NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

19980727 170

THESIS

IMPLEMENTATION OF A MULTIPLE ROBOT
FRONTIER-BASED EXPLORATION SYSTEM AS A
TESTBED FOR BATTLEFIELD RECONNAISSANCE
SUPPORT

by

Patrick A. Hillmeyer

June 1998

Thesis Advisor: Xiaoping Yun

Approved for public release; distribution is unlimited.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 1998	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE IMPLEMENTATION OF A MULTIPLE ROBOT FRONTIER-BASED EXPLORATION SYSTEM AS A TESTBED FOR BATTLEFIELD RECONNAISSANCE SUPPORT	5. FUNDING NUMBERS

6. AUTHOR(S)
Hillmeyer, Patrick A.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Space and Naval Warfare Systems Center – San Diego San Diego, CA 92152-5001	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Future military battlefields will see smaller forces responsible for ever increasing geographical areas. In addition, future conflicts will occur more often in urban or built-up areas. Both of these trends argue for some type of augmentation for initial reconnaissance, continued observation, and control of lines of communication and other key terrain features. Multisensor systems, mounted on a variety of robotic platforms, can provide this type of battlefield support where it is needed most. However, before costly decisions concerning the details of such systems can be made, basic research needs to be conducted regarding their most effective composition and utilization.

Prior to this time all multiple robot studies at this institution had only taken place in simulated environments. This thesis implements a real-world multiple robot system that uses a technique known as frontier-based exploration to explore and map a laboratory or office environment. In doing so, many previously hidden aspects of multiple robot systems, unnoticeable in simulation-only studies, become evident. The results developed here are compared to results obtained elsewhere involving other robotic platforms. This research lays the foundation for future research involving multiple robots interacting as a system in a real-world environment and acting towards a common or shared goal.

14. SUBJECT TERMS Robotics, Multiple Robots, Sensor Fusion, Battlefield Reconnaissance		15. NUMBER OF PAGES 331
		16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

IMPLEMENTATION OF A MULTIPLE ROBOT FRONTIER-BASED EXPLORATION SYSTEM AS A TESTBED FOR BATTLEFIELD RECONNAISSANCE SUPPORT

Patrick A. Hillmeyer
Captain, United States Marine Corps
B.S., University of Minnesota, 1990

Submitted in partial fulfillment
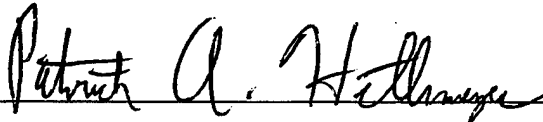of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
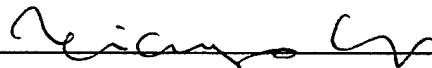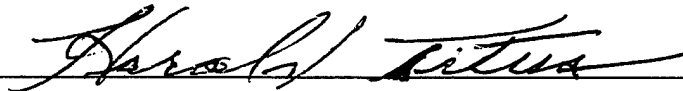**June 1998**

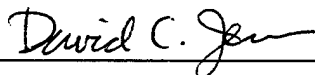Author: _____
Patrick A. Hillmeyer

Approved by: _____
Xiaoping Yun, Thesis Advisor

_____
Harold Titus, Second Reader

_____
for Herschel H. Loomis, Jr., Chairman
Department of Electrical and Computer Engineering

iii

ABSTRACT

Future military battlefields will see smaller forces responsible for ever increasing geographical areas. In addition, future conflicts will occur more often in urban or built-up areas. Both of these trends argue for some type of augmentation for initial reconnaissance, continued observation, and control of lines of communication and other key terrain features. Multisensor systems, mounted on a variety of robotic platforms, can provide this type of battlefield support where it is needed most. However, before costly decisions concerning the details of such systems can be made, basic research needs to be conducted regarding their most effective composition and utilization.

Prior to this time all multiple robot studies at this institution had only taken place in simulated environments. This thesis implements a real-world multiple robot system that uses a technique known as frontier-based exploration to explore and map a laboratory or office environment. In doing so, many previously hidden aspects of multiple robot systems, unnoticeable in simulation-only studies, become evident. The results developed here are compared to results obtained elsewhere involving other robotic platforms. This research lays the foundation for future research involving multiple robots interacting as a system in a real-world environment and acting towards a common or shared goal.

TABLE OF CONTENTS

x

ACKNOWLEDGEMENTS

I. INTRODUCTION

A. GENERAL

As the general downsizing of the military continues, the trend on future

battlefields will be toward smaller units being responsible for scouting and securing larger

areas. It is also predicted that, in the near future, 70 percent of the world's population

will be in urban areas [Ref. 1]. As more and more of the general population moves into

cities, future battlefields are more likely to be in urban or built-up areas. As these trends

continue the need for some type of robotic support and augmentation for the small unit or

individual on the ground will become greater.

Urban and built-up areas present some of the greatest challenges for military units

in the areas of initial reconnaissance, continued observation, and control of lines of

communication and other key terrain features. Multisensor systems, mounted on a

variety of robotic platforms, can provide this type of battlefield support in areas where it

is needed most. However, before making very costly decisions about the make-up of

these systems it is imperative to conduct some basic research about the types of systems

that are most cost effective and most efficient. This will allow the system designers to

make intelligent decisions about the type and composition of systems that will be most

useful on tomorrow's battlefield.

B. PROBLEM STATEMENT

As a reconnaissance system is designed there are several fundamental questions that must be asked and answered. In some missions, a large number of the simplest possible systems may be the right answer. This might be the case if all that is desired is simple detection of "something" with no detail other than the fact that there is "something" in the vicinity of the systems sensors. For other missions the best solution may be a smaller number of systems incorporating higher capability sensors, increased processing capability, improved communications resources, and greater mobility. This would be the case if the system were required to perform more complex tasks such as target identification, target tracking, or even target attack. Or perhaps the best system for the mission lies somewhere in-between these two extremes or is even a combination of them both. [Ref. 2]

This thesis will explore a comparison between the first and second options. A robotic mapping system was originally developed for a small number of very expensive, but very sensor capable, NOMAD 200 robots. By taking this same system and implementing it on a larger number of much less expensive, but less capable, NOMAD SCOUT robots the beginnings of a comparison between the two options will be possible.

In addition, many of the challenges and questions inherent to the development of a multiple robot mapping system are also present in any research involving multiple robots attempting to accomplish a common task. The problems of communication, coordination, and control apply similarly to multiple robot mine clearing systems and robotic weapon

2

platforms. It is hoped that the development of a multiple robot testbed can set the stage

for further research in these areas at this institution.

C.   OUTLINE OF THE THESIS

Chapter II provides a description of the platforms and sensors that were used for

the research in this thesis, as well as the platforms used in previous work for comparison

purposes. Chapter III discusses the other hardware and software common to this and

other research that was used to provide connectivity and control within the systems.

Chapter IV provides background on the techniques of robotic map building. Chapter V

describes the exploration strategies and techniques used in this and other studies. Chapter

VI describes the methods of integrating the work of multiple robots in a cooperative map

making effort. Chapter VII presents the results that were obtained at the Naval

Postgraduate School (NPS) based upon the system originally designed at the Naval

Research Laboratory (NRL). Finally, Chapter VIII discusses the conclusions and

recommendations for follow-on studies.

4

II.   PLATFORMS AND SENSORS

This chapter provides background information so that the reader has an understanding of the NOMAD 200 and NOMAD SCOUT mobile robots and the similarities and differences between these two platforms. It will also describe the sensors available on each of the platforms, as well as some details concerning previous work done with the NOMAD 200 at both NPS and NRL. Figure 1 provides a relative size and shape comparison of the NOMAD 200 and NOMAD SCOUT.

Figure 1. Relative size and shape comparison of the NOMAD SCOUT (left) and NOMAD 200 (right) – note telephone book in front of NOMAD 200 for reference.

A.   NOMAD 200 MOBILE ROBOT

The NOMAD 200 is an integrated mobile robot system with four sensory modules including tactile, infrared, sonar and laser sensors. There are multiple onboard computers that provide sensor and motor control, as well as providing communication to the host computer via a wireless Ethernet system [Ref. 3]. Figure 2 provides a detailed picture of the NOMAD 200.

1.   Mechanical Description

The NOMAD 200 base chassis is driven by a three-wheel synchronous drive mechanism, using one motor to drive all of the wheels and a second motor to steer all of the wheels. The robot has a zero gyro-radius, meaning that it can rotate about its own center. It can translate at up to 24 inches per second and rotate at a maximum rate of 60° per second. The base is 18 inches in diameter, which extends to 21 inches with the bumper installed. The NOMAD 200 stands 31 inches tall, excluding any additional sensors added on top of the robot [Ref. 4]. The turret (which all the sensor systems are mounted on) can be rotated independently of the base. [Ref. 4]

2.   Sensor Systems

The basic NOMAD 200 sensor array includes tactile (bumper) sensors, infrared sensors, sonar sensors, and laser sensors. In addition to these sensor systems, the robot also has an odometric system that tracks the robot's movements. The encoder resolution

on this odometric system is 18 counts/cm for translation and 1510 counts/degree for robot

steering and movement of the turret.

Figure 2. Detailed close-up picture of the NOMAD 200.

The tactile system consists of a bumper ring positioned over 20 independent pressure-sensitive sensors. These simple on-off sensors are interleaved around the bumper ring in order to provide 360° coverage with 18° resolution. [Ref. 4]

The infrared sensors aboard the NOMAD 200 incorporate a 16 channel, reflective intensity based, infrared ranging system that provides 360° of coverage. Each of the 16 sensors is composed of two light emitting diode (LED) emitters and a photodiode detector enclosed in a delrin housing. The range from the sensor to the object(s) is determined by the amount of light from the emitters that is reflected back to the detectors after striking the object(s). The reflectivity of the object greatly affects the reading. Thus the system needs to be calibrated for the environment in which it is to be used. With appropriate calibration, range accuracy is within 5% from 0 to 24 inches from the sensor. [Ref. 5]

The sonar sensors on the robot are composed of a 16 channel, time-of-flight based, sonar ranging system. The system uses standard Polaroid transducers. Each transducer has a beam width of 25°. Range from the sensor to object(s) is determined by the time of flight of the acoustic signal generated by the transducer and reflected back by the object(s). The user has the option to control the firing sequence of the 16 individual sonar sensors mounted about the circumference of the robot. To minimize the potential for sensor interaction, a non-sequential firing sequence is recommended. [Ref. 6]

The laser sensor on the NOMAD 200 is a two-dimensional, triangulation-based laser ranging system. A laser diode is used as the light source and a charged-coupled device (CCD) array camera is used to generate an image. The laser diode produces a

horizontal "plane" of light. The CCD camera is placed vertically above this "plane" and inclined downward. Any object intersecting this plane forms a light stripe on the image generated by the CCD camera. The range to this object is found by determining the position of this light stripe along the scan lines of the camera. This system has an operating range from 12 to 120 inches. [Ref. 7]

3.   Previous Work with this Platform

The NOMAD 200 has been and continues to be a very popular research platform at NPS and elsewhere. There exists an extensive body of work involving the NOMAD 200 in several areas in the field of robotics. Some of the more recent work at NPS has involved localization of the robot position in an unknown environment [Ref. 8, 9] and geometric formation and movement in formation of multiple robots in simulation [Ref. 3]. Because NPS only has a single NOMAD 200 (due primarily to the expense of the platform), all work involving multiple robots had to be simulated until very recently. This single robot limitation coupled with the high logistics cost of setting up a very complicated robot platform have been major factors in limiting research performed with real, vice simulated, robots at NPS.

NRL has also done extensive research using the NOMAD 200. Their acquisition of two NOMAD 200 robots has allowed them to conduct more actual research involving multiple robots in addition to simulations. The basis for this thesis is an adaptation of some of their work (described below) in order to form a testbed for actual multiple robot work here at NPS involving less costly platforms.

9

B.   NOMAD SCOUT MOBILE ROBOT

The NOMAD SCOUT is an integrated mobile robot system with ultrasonic and tactile sensors, as well as an odometric system. It uses a multiprocessor, low-level control system that controls the sensing, motion, and communications. At a high level, the SCOUT is controlled either by a laptop, mounted on top, or a remote workstation communicating via radio modem. The SCOUT is code compatible with NOMAD 200 class robots [Ref. 10], which was a very important consideration in its choice as a research platform at NPS. Figure 3 provides a detailed picture of the NOMAD SCOUT.

1.   Mechanical Description

The NOMAD SCOUT is a two-degree of freedom (DOF) differential drive robot. The drive is set about the geometric center of the robot, which allows the robot to turn or rotate about its own axis. The NOMAD SCOUT has a maximum speed of one meter per second with a maximum acceleration of two meters per second squared. The robot is .38 meters in diameter and is .34 meters in height. Without batteries, the unit weighs 23 kilograms. [Ref. 11]

2.   Sensor Systems

The basic NOMAD SCOUT sensor array includes tactile (bumper) sensors and sonar sensors. In addition to these sensor systems, the NOMAD SCOUT also has an odometric system that tracks the robot's movements. The encoder resolution on this

odometric system is 167 counts/cm for translation and 45 counts/degree for robot

rotation.

Figure 3. Detailed close-up picture of the NOMAD SCOUT with radio modem.

The tactile system uses a ribbon switch enclosed in an energy absorbing neoprene channel to provide 360-degree coverage. The ultrasonic system uses 16 independent sonar sensors. The effective range of the ultrasonic sensors is 6 to 255 inches. This sonar sensor is basically identical to the one installed in the NOMAD 200 with slight differences due to the smaller diameter of the NOMAD SCOUT. [Ref. 11]

C.   COMMUNICATION AND COMPUTATIONAL RESOURCES

No description of the platform would be complete without mention of the software and hardware that allow the robots to operate. A thorough knowledge of the underlying hardware and software is essential in understanding the model being used in this research.

1.   Software

The Nomadic Host Development Environment (NHDE) is a full-featured, object-based mobile robot software development package for both the NOMAD 200 and NOMAD SCOUT mobile robots [Ref. 12]. The complete package provides the control and graphics interfaces as well as a very realistic simulation tool for program testing. By using the supplied development package it is much easier to concentrate research on higher level issues of motion planning and control because most of the lower level issues of sensor and motor control are handled by the included software.

The control interface allows for programming the NOMAD 200 or NOMAD SCOUT using a high-level programming language (either C, C++ or Lisp) and linking to a supplied library [Ref. 12]. Built-in to the supplied library are interfaces to the supplied driver software that handle lower level functions such as sensor and motor control. This allows for a higher level of abstraction in the researcher's approach. The graphical user interface and simulator are accessible when the user runs the executable server program, *Nserver* [Ref. 3].

The graphical user interface in the NHDE is based on the OSF/Motif graphics toolkit for the X Window System [Ref. 12]. The graphical interface can display information on up to six robots simultaneously. There are several different windows displayed in the complete graphical user interface. First, there is the world (map) window which gives an overall view of the environment (real or simulated) that the robots are in, as well as the positions of the robots relative to the environment and one another. Secondly, there is a robot window, with one copy per robot, which contains information about each individual robot. This information includes the current command being executed, position and orientation information, and sensor information. Along with each robot window, there are two more windows that give more detailed information about current sensor readings. These two windows are usually used to display a graphical representation of the sonar and infrared returns of each robot's sensors. Detailed information about each robot can be saved as a setup file (*robot.setup*) [Ref. 3]. This includes information about the model of robot (NOMAD 200 or NOMAD SCOUT) being used.

The Nomadic Simulator is a fully functional mobile robot simulator that can accurately model most environments, the robot's motion, and its sensing capabilities. There is a high degree of correlation between the simulated world and the real world. If a program will not run on the simulator it definitely will not work on a real robot. The simulation tool allows the researcher to build a controlled environment in which to develop and debug programs. Using a graphical drawing tool a user can draw a map in the world window to simulate the desired surroundings. This file can be saved as a setup file for the world (*world.setup*). In addition, once a program is running properly on a simulated robot, it can be switched to a real robot via a pull-down menu within the simulator. Once this is done the graphical interface will then begin to display information from the real robot vice the simulated one while commands from the program will control the real robot via the server program (*Nserver*). [Ref. 3]

The NHDE also incorporates several other very convenient features that aid the researcher. There is a record and playback tool that that allows for sensor data and/or executed commands to be stored for later analysis, as well as providing for an instant replay capability. This is an invaluable debugging tool. There is also a console available on *Nserver* that allows the user to directly input to the robot any possible command. This is a very handy option for checking the robot's sensors or making small, subtle adjustments in the robots location during experiments. In addition, there is an on-screen, software joystick that allows the user to remotely drive the robot. This is often used to move actual robots around in the real world while simultaneously collecting sensor data.

This allows the researcher to write software that collects and manipulates sensor data without having to also write code to handle the robot's motion.

Figure 4 shows a typical NHDE display from an experiment involving two robots in a simulated environment. Also shown in the figure are many of the features described above such as the global map window, the robot window with its associated sensor windows, the record and playback window, the command console window, and the software joystick window. Also demonstrated in the robot window is another feature available in NHDE. The user has the option to display raw sensor return data or "hits" in the robot window as well as a copy of the global map. This provides a quick visual reference to the researcher indicating whether or not the robot's sensors are functioning properly.

Using *Nserver* in conjunction with the graphical interface and simulation tool is a very convenient way to test and debug software in simulation for subsequent use on a real robot. However, because of the client-server architecture, testing client programs on the real robot via *Nserver* may slow the control and data return rates because *Nserver* acts as a router. Once a program is working properly it can be recompiled to use the control interface library directly without the need for *Nserver* to be running concurrently. This is a very simple process because of the efficient design of the NHDE software.

Each of the application programs for each robot, as well as the *Nserver* when used, can run simultaneously as separate processes under the UNIX operating system. All communications between the host processes that are controlling the robots are

handled as communications between UNIX processes using the TCP/IP protocols and a server-client architecture. This will be described in more detail in Chapter IV.

Figure 4. Typical graphical display. (From Ref. [12])

2.   Hardware

As mentioned above, each program runs as a separate UNIX process. Whenever practical each process is run on a separate Sun workstation because the frontier-based exploration program is computationally demanding. In addition, running each robot with a separate workstation is more faithful to the system that is being modeled as described in Chapter IV.

16

Communications from host process to host process are handled as described

above and in more detail in Chapter IV. Communications between the host process and

the robot it controls are handled via radio Ethernet. Each robot has a 2.4 GHz radio

modem. This radio modem is assigned an IP address that the host process uses to route

instructions to the robot [Ref. 13]. The radio modem connects to a wireless access point

that provides connectivity between the radio modem and the rest of the network [Ref.

14].

III. EVIDENCE GRID BASED MAPPING TECHNIQUES

Over the years many methods have been developed to convert sensor data from robots into useful maps. At times it seems that there are as many different types of robotic mapping techniques as there are robots. Each group of researchers has approached the problem with a slightly different variation or procedure. However, one major method that has proven very successful is called the "grid method."

A. OVERVIEW OF GEOMETRIC MAPPING TECHNIQUES

A geometric map represents objects according to their geometric relationships. It can be a grid map, or a more abstracted map, such as a line map or a polygon map [Ref. 15]. A geometric map also has the advantage of being easily interpreted by humans trying to match the map with the area that it represents. The key is finding the method that builds the best map.

One of the problems with building a map using simple lines and polygons is that the mapping capability breaks down quickly in a non-simulated environment. These methods depend on interpreting small amounts of sensor return data and mapping points, lines, or surfaces to that data. This can work very well in a simulated environment where obstacles tend to be composed of simple geometric shapes and straight lines, but the real world is not made up of such convenient shapes. Often times such methods map false obstacles or incorrectly shaped obstacles based on extraneous or incorrect sensor data. This is especially a problem when dealing with sensors that return much "noisy" data by

19

their nature, as do sonar sensors. So these line and polygon mapping techniques are said to be lacking robustness.

The grid technique was developed as a way to overcome many of the problems described above. When using a grid method it is not necessary to make assumptions about the shape or size of an object being mapped. Simply plotting enough sensor return points on a grid forms a recognizable map that can be used by both robots and humans. Once enough points are plotted, edge detection techniques can then be used to pick out walls, obstacles, unexplored areas, and other terrain features. More about edge detection techniques and their uses in mapping will be discussed in Chapter IV.

B. SIMPLE PLOTTING OF SENSOR DATA

Perhaps the most basic of the grid methods is simply plotting the data returned by the robot's sensors and marking areas within the sensors' range as either occupied or unoccupied. This approach has the advantage of simplicity and can produce a reasonably good map in a well-defined simulated environment. However, in a real world environment using only sonar as a sensor this method quickly breaks down unless very tight constraints are set on the range of data used. This was the first mapping technique attempted for this research in conjunction with a single robot. Although this method was later abandoned in favor of a technique better suited for a multiple robot system, it is still useful in depicting some of the complexities of robotic mapping systems.

Figures 5 and 6 illustrate a comparison of simulated and real-world maps constructed by the simple plotting of sonar return data with varying range limitations

20

imposed on the displayed data. Figure 5(a) is a typical simulated environment used to test various robotic map-making techniques. The environment is a roughly 325 by 275 inch rectangle with several large geometrically shaped obstacles placed within it. Figure 6(a) is a photo of an aisle between two laboratory benches that was used to test map-making methods in a real environment. The aisle has several chairs with metal legs along the robot's projected path as well as open spaces beneath the benches. In both the simulated and real worlds the robot is remotely moved through the environment via the software joystick described above.

At the same time the user remotely moves the robot, another process is running which collects the sonar return data and the robot position and orientation (*pose*) data and writes it to a data file. Pose data is very important in converting the sonar returns for a given robot position and orientation into data that can be mapped onto a common coordinate system. After the data were collected, a MATLAB routine read the data file, transformed the sonar return and pose data, and plotted the resulting map. Along with the sonar return data the robot's path in the simulated or real world is also plotted as a dotted line in the resulting map. In this case the map was generated after maneuvering the robot, but *Nserver* also allows for the raw sensor returns to be plotted in real time within the robot window.

In Figures 5(b) and 6(b) the reliable sonar range is set to 255 inches (the maximum rated reliable range according to the manufacturer's specifications). Thus, the mapping program plots all sonar return data that is below 255 inches. A return of 255 inches is regarded as open space and not plotted. In the simulated world this produces a relatively

good map with some noise at corners and other line intersections. In the real world there is much noise and what appear to be many extraneous returns. This noise and the apparent false returns will be discussed in more detail in Chapter IV.

In Figures 5(c-e) and 6(c-e) the reliable sonar range is steadily reduced and a larger percentage of the raw sonar returns eliminated and not plotted. Correspondingly, as the outlying returns are discarded, the data that is plotted produces clearer and less noisy maps. Unfortunately, as can be seen very well in Figure 5(e), dropping the longer returns in the larger simulated environment resulted in the robot path being too distant from several obstacle walls, preventing them from being mapped. This illustrates the tradeoff between reducing the reliable sonar return range in order to get quality data, and forcing the robot to travel further in order to close in on all mappable objects in the environment. More about this tradeoff will be discussed in greater detail in Chapter V.

Plotting every sensor return does not prove to be the best method of constructing a useable map. The same simplicity that makes it so easy to implement also proves to be its downfall in real world situations. It is possible to fuse maps together with this method by converting all returns to a common coordinate system, but the main problem is that all data is given the same amount of validity. What is needed is a method to weigh, or measure, the "goodness" of sensor data from multiple sensors at multiple positions and build a map accordingly.

Figure 5. Illustration of simulated environment and the resulting sonar maps formed by maneuvering the simulated robot throughout it, collecting sonar range data, and plotting subsets of that range data based on estimated reliability.

Figure 6. Illustration of real world environment and the resulting sonar maps formed by maneuvering the actual robot throughout it, collecting sonar range data, and plotting subsets of that range data based on estimated reliability.

24

C.   THE EVIDENCE GRID METHOD

The evidence grid method was developed as a technique to create high resolution maps from wide-angle sonar. This approach allows range measurements from multiple points of view to be systematically integrated into a common map. The integration technique allows for multiple readings of an area to either reinforce or refute one another as to the whether or not the area is occupied. As more sensor data is added the definition of the map improves. [Ref. 16]

In the evidence grid method, the area to be mapped is divided into a grid with M X N cells. In each of these cells is stored a set of information regarding the best estimate as to what that cell contains. This "evidence" can be many different things, such as the surface orientation or color of whatever is in the cell, but for mapping perhaps the most useful information is occupancy information concerning the cell [Ref. 17]. Early versions of this method [Ref. 16, 18] stored this occupancy information as a two part record with a status of either *unknown*, *empty*, or *occupied* and an associated certainty factor of either 0, -1 to 0, or 0 to 1 respectively. Early versions also used ad hoc formulas to combine this information about each cell into a useable map.

Later implementations [Ref. 17, 19, 20] eliminated the two part occupancy record and replaced it with a single value representing the probability that the cell is occupied. This technique also allowed for a better method of combining and integrating sensor data using a variation of Bayes theorem. In this representation, an unknown or unexplored cell would have an occupancy probability of 0.5. As more sensor data becomes available this

25

probability changes accordingly. The details of how this sensor data is integrated together are the topic of the next section.

D. FUSING SENSOR DATA USING AN EVIDENCE GRID

The major advantages of the evidence grid method over previous methods are the ability to weigh or measure the "goodness" of the sensor data and the ability to fuse this data in a simple, yet effective, manner. The best way to demonstrate how the sensor data is fused is to first present a graphical representation and then go more in depth into the mathematics behind it.

1. Graphical Presentation

Figure 7 shows a sequence of images simulating the fusion of sonar sensor data from two different points. In this sequence, the circles labeled A and B represent locations at which a sonar sensor sends out a pulse and receives a return. Points A and B could be different sensors on the same robot, the same sensor on a single robot at a different time, location and/or orientation, or two different sensors on two separate robots. The advantage of this sensor fusion method is that for all practical purposes, exactly what they are does not matter. The only thing that matters is that the readings be relatively independent of one another. For the purposes of this explanation, they will be referred to as sensor A and sensor B. The background grid will represent the evidence grid that will be developed in order to map the region. As per most recent implementations of

the evidence grid model it will be assumed that all the cells on the grid will be initialized to an initial occupancy probability of 0.5.

In Figure 7(a) the relative geometric locations of A and B are shown, along with some object in the distance. Figure 7(b) shows a two-dimensional "slice" of a three-dimensional sonar cone emanating from A. R_{maxA} is the maximum effective range of sensor A. R_{objA} is the range at which the object is detected some distance away from A. Because of the wide-angle nature of the sonar sensor the object is known to be only somewhere on a certain surface [Ref. 20]. In this case the dashed line represents the edge of the circular "slice" of the sonar cone along which the object might lie. So at this point it is known that an object lies somewhere within a constrained region.

The shape of that region and its distance from A are also known, but there is not enough information at this point to make an assumption about exactly where in that region the obstacle exists. Under the evidence grid model the occupancy probabilities of the cells along that entire region would be increased. In the two-dimensional case this would mean all the cells along the dashed line. The amount of increase might vary depending on several factors such as distance from target, angle from sensor, or other measures of sensor data quality, but the main point is that the occupancy probabilities along the assumed obstacle location would be increased relative to the surrounding cells.

Figure 7(c) shows the two-dimensional representation of the sonar cone emanating from B. Again, B may be the same sensor as A at different time, location, and/or orientation or a separate sensor on the same or a different robot. In this case B does not sense the obstacle within the region it is observing.

(a)

(b)

(c)

28

(d)

(e)

Figure 7. Example of fusion of sonar return data from two geographically different sonar sensors.

Figure 7(d) displays both the sensor readings from A and B simultaneously. In this case part of the sensor reading from B overlaps some of the region of the sensor reading from A where the cells had had their occupancy probability raised relative to the surrounding cells. However, the data from sensor B now provides additional information

29

concerning portions of this area and that information can be used to adjust the occupancy probabilities in that region accordingly.

Figure 7(e) shows an enlarged view of the area where the sensor readings from *A* and *B* intersect. In the evidence grid model the information from *A* and *B* would be combined in such a way as to lower the occupancy probabilities of the cells within the circle marked *Region 1* that are in view from both *A* and *B*. At first it would seem logical to also increase the occupancy probabilities of those cells within the circle marked *Region 2* as well. However, because the evidence grid model used considers each sensor reading to be relatively independent of every other sensor reading, the reading from *B* cannot be used to adjust the occupancy probabilities of the cells in *Region 2* because those cells are not in view of *B*.

Depending on the detailed specifics of the model chosen, the occupancy probabilities of the cells in *Region 1* might still be higher than their immediate neighbors that were always considered empty, but they would always be lower than those of the cells in *Region 2*. So even though the cells in *Region 2* are not directly changed through this process, their occupancy values are now the local maxima for the overall area within the large evidence grid shown in the figure.

Figure 7 illustrates the evidence grid method on a small scale. Now imagine it on a much larger scale with multiple sensors on multiple platforms and many hundreds to thousands of relatively independent sensor readings from a multitude of ranges and orientations. It is by combining all of these together that a map is created using the evidence grid method. The mathematical details of this process are described below.

2.   Mathematical Presentation

Perhaps the clearest and most concise mathematical description of the evidence grid method of combining sensor data can be found in [Ref. 17]. What follows in this section is a condensed version of that work presented here for clarity and completeness.

Let $p(A|B)$ represent the best estimate of the likelihood of situation A given that information B has been received. A and B mean either "a certain region of space is occupied," (written o), "a certain region of space is unoccupied", (written $\bar{o}$), or they represent a sensor reading. By definition, $p(A|B) = p(A \cap B)/p(B)$. The quantity $p(A)$ represents the estimate of A given no new information. The alternative to situation A is written $\bar{A}$, (read as "not A").

For the two occupancy cases of a cell, o (the cell is occupied) and $\bar{o}$ (the cell is empty), and new information M (derived from a sensor measurement), the above definition creates the equation:

$$\frac{p(o|M)}{p(\bar{o}|M)} = \frac{p(M|o)}{p(M|\bar{o})} \frac{p(o)}{p(\bar{o})} \tag{1}$$

Now this can be rewritten as:

$$\frac{p(M|o)}{p(M|\bar{o})} = \frac{p(\bar{o})}{p(o)} \frac{p(o|M)}{p(\bar{o}|M)} \tag{1a}$$

Now suppose that there exists some information, M_1, that has already been processed into a map, i.e. $p(o|M_1)$ already exists and it is desired to integrate some new measurement, M_2, to find $p(o|M_1 \cap M_2)$. In order to make the analysis tractable it is assumed that the new measurement is independent from all previous information. This

may not be completely true, but for the purposes of constructing a map from many sensor inputs it simplifies the problem immensely. However, it is not implied that $p(M_1 \cap M_2) = p(M_1)p(M_2)$, since if M_1 indicates that the cell is occupied then it is hoped that M_2 would be more likely to indicate the same thing. Instead, what is meant is that, given that the cell is occupied, the probability of getting reading M_1 is independent of getting M_2, and similarly for the cell being occupied:

$$p(M_1 \cap M_2 | o) = p(M_1 | o)p(M_2 | o) \tag{2}$$

$$p(M_1 \cap M_2 | \bar{o}) = p(M_1 | \bar{o})p(M_2 | \bar{o}) \tag{3}$$

Another way to look at this assumption is that it is only assumed that the sensor's errors are independent from one reading to the next. This is especially true of noisy sonar sensor data, in which the errors vary greatly from one reading to another from the same sensor due to changes in range, orientation, etc. Combining this assumption with a single application of Equation 1a, results in:

$$\frac{p(o | M_1 \cap M_2)}{p(\bar{o} | M_1 \cap M_2)} = \frac{p(o | M_1)p(M_2 | o)}{p(\bar{o} | M_1)p(M_2 | \bar{o})} = \frac{p(o | M_1)p(o | M_2)p(\bar{o})}{p(\bar{o} | M_1)p(\bar{o} | M_2)p(o)} \tag{4}$$

We generally assume that the a priori probability of a cell being occupied is 0.5, i.e., $p(o)=p(\bar{o})=0.5$, so that the last factor in Equation 4 cancels out. When the information M_2 is a sensor reading, the value $p(M_2 | o)/p(M_2 | \bar{o})$, for all cells and all possible readings, is called the sensor model. In other words, the sensor model is a function which attaches a number, $(p(M_2 | o)/p(M_2 | \bar{o}))$, to every combination of sensor reading and cell location, relative to the sensor. This assumes that the sensor is isotropic in its world position and pointing direction. In general, the sensor model is a

function of the sensor reading, the location and orientation of the sensor, and which cell is being updated. Also, while the sensor reading M_2 may represent a continuous number indicating distance from the sensor, in general, each time the sensor is polled it will return an element from some set and M_2 will range over all elements of that set.

The sensor model is usually independent of the current map and can be stored in tables. A further speed up of the process can be achieved if the logarithm of the above probability ratio is used. In this case the model uses:

$$log \frac{p(o \mid M_1 \cap M_2)}{p(\overline{o} \mid M_1 \cap M_2)} = log \frac{p(o \mid M_1)}{p(\overline{o} \mid M_1)} + log \frac{p(o \mid M_2)}{p(\overline{o} \mid M_2)} \qquad (5)$$

In the logarithmic method the combining formula is changed from a multiplication to a simple addition. In this case the logarithmic result itself can be considered as weight of evidence of cell occupancy. Therefore, the need for only a single addition per cell allows for very rapid updating of the evidence grid map based on the newly acquired sensor data.

IV.   FRONTIER-BASED EXPLORATION

One of the goals of a robotic-based reconnaissance system is to reduce the amount of manpower required to reconnoiter an area. Any system worth building should be able to explore at least a limited area autonomously and in a fairly efficient manner. This means that the robots will have to be able to make use of the maps they will build as they move through their environment. Chapter III already discussed in great detail the way those maps might be represented, now it is time to discuss how a robot would use them to explore and map an area on its own.

A.   DEFINITION

A robot exploring a new area will initially know nothing about that area except what it can detect in its immediate area with its own sensors. The limits of its sensors will form a boundary between known, explored space and unknown, unexplored space. Such a boundary is called a frontier. Within this boundary it is assumed that all obstacles are known and mapped. Thus, further exploration within this boundary would be futile. In order to maximize exploration in the least amount of time, the robot should move to the boundary between explored and unexplored space as soon as possible and use its sensors to expand the explored region. At the same time the act of expanding this known region will in turn create new boundaries or frontiers for the robot to explore. This is the central idea behind frontier-based exploration. This process is illustrated in Figure 8. [Ref. 22]

Figure 8. Example of frontier-based exploration. Robot begins by scanning immediate area, incorporates result into its map noting frontiers yet to be explored, moves to a frontier, and repeats the process.

Figure 8(a) shows a robot at startup in some unknown environment. The black circle represents the robot. The two large black rectangles represent obstacles in the area to be explored. The white space surrounding the robot is the area it can directly detect with its own sensors. The gray background represents the unexplored regions and the black dashed line represents the boundary between the known and unknown space.

In Figure 8(a) the robot can detect the top edge of the first obstacle, but the obstacle blocks its view what may be behind the obstacle. Accordingly, the robot chooses a frontier and proceeds to that frontier in order to continue the process of exploration and map building. Figure 8(b) shows the robot in its new position now able to see behind the first obstacle. After scanning in this position the robot moves on to the position shown in Figure 8(c) where it can now detect the second obstacle. This process could continue indefinitely as long as there are unexplored frontiers for the robot to explore.

B. FRONTIER DETECTION

Before a robot can decide on a frontier towards which to proceed in order to continue exploration, it must first decide where the frontiers are within the region it has already explored and mapped. In order to detect these frontiers a process similar to edge detection and region extraction in computer vision (also known as machine vision) is used to find the boundary between mapped open space and unmapped unknown space [Ref. 23].

Over the years many techniques have been developed in the area of computer vision to extract information from photographic images based on the pattern of changes in brightness in the picture [Ref 24]. Most of these techniques involve decomposing the image into a set of pixels, much like a grid. Each cell in the grid is given a value based on the brightness of the pixel that the cell represents from the original image. This grid is then searched for patterns that may indicate edges or patterns of interest.

There are a variety of methods for picking out the patterns in the image depending on the information that is desired. Some techniques involve the use of a set of "masks" composed of a small (on the order of 3 X 3 or 4 X 4) pattern of cells of varying values that are successively laid across the original image. As the mask is moved across the image the cells of the mask are convolved with the cells of the image beneath the mask and the resulting matrix indicated the presence of edges in that portion of the original image. Other methods rely on simple histograms of the brightness values of the cells in the original image and attempt to use curve-fitting techniques to pick out edges of objects in the real world.

Regardless of the methods used, these same types of techniques can be applied to detecting boundaries in frontier-based exploration. The same underlying grid format will be used as in computer vision, but in the case of frontier detection the values in the cells do not represent the brightness of a pixel. Rather, they represent the occupancy probability of a cell in the area the robot is exploring.

In general each cell in the area being examined for frontiers will be placed into one of three categories [Ref. 23]:

- **open:** when the current occupancy probability of the cell is less than the prior probability

- **unknown:** when the current occupancy probability of the cell is the same as the prior probability

- **occupied:** when the current occupancy probability of the cell is greater than the prior probability

A short explanation about the term "prior probability" is needed. When the evidence grid is first created it is necessary to initialize the cells in the grid to some value. Since at creation nothing in the grid has been explored it is logical to initialize all the cells to an occupancy probability value representing unknown, unexplored areas. All the implementations of frontier-based exploration discussed here [Ref. 22, 23, 25] initially set the cells' values to 0.5. When the frontier-detection process is first done it is this initial (prior) occupancy probability to which the current occupancy probability will be compared.

After a robot starts up or moves to new frontier it will make a sensor scan of the surrounding area. Based on the information returned from its sensors it will updated the occupancy probability value of any cell with direct range of its sensors. The robot will then use this updated information to perform frontier detection. Any open cell adjacent to an unknown cell will be labeled as a frontier edge cell. Adjacent edge cells are then grouped into frontier regions. Any frontier region above a certain minimum size (say

roughly the size of the robot) will be considered an accessible frontier and marked as such. [Ref. 23]

C.   NAVIGATION

Once the robot has scanned an area and updated all cells of the evidence grid within range, it must now safely and efficiently navigate to a new frontier in order to continue exploration and map building.

1.   Route Planning

Navigation to a new frontier should involve a path that is completely within known space, therefore, all obstacles in that space should be known. Route planning involves choosing the best path (based on some criterion such as length of path, nearest approach to obstacle, etc.) through the known space to the frontier to be explored. This path will be based on the latest update of information concerning explored space.

There are various algorithms such as depth-first, breadth-first, and A* search routines [Ref. 26] that attempt to search through a known map for safe and efficient navigation paths. However, a problem can arise if an obstacle (for example, another robot) moves into the chosen path since the last time information about the known space was updated. This may lead to the path the robot chooses to move to a new frontier being blocked. In that case, in order to avoid collision with the new obstacle and continue

navigating to the chosen frontier requires that the robot have some means of reactive avoidance.

2. Reactive Obstacle Avoidance

Reactive obstacle avoidance entails some method of detecting and reacting to mobile obstacles that appear in the robot's path that were not in the then-current map used by the route planning routine to plot a safe path for the robot. Mobile obstacles may include humans, other robots, or any of a number of other unpredictable phenomena that may exist in the robot's world. For the most part, reactive obstacle avoidance involves the use of relatively short-range sensors (IR, contact, etc.) or long-range sensors (sonar, vision, etc.) scanning in the area immediately surround the robot as it moves.

One important note about sensors and reactive obstacle avoidance is that the required scanning rate of the obstacle avoidance sensors is closely related to the expected travel rate of the robot and the anticipated characteristics of the area in which it is traveling. Obviously a quickly moving robot in an area where new obstacles appear frequently will need to scan the local area around it much more frequently than a slow moving robot in a well known, stable area.

There are several different actions that a robot might take upon detecting a new obstacle along its planned path. One common method is to use some sort of very low-level navigation routine that simply finds a path around the new obstacle and gets the robot back on the previous planned path as quickly as possible. This eliminates the need to call on the slower full route-planning algorithm. For small obstacles in the robot's

41

planned path this is usually the method used. However, a problem can arise with this method when the new obstacle is so large that the low-level process cannot easily find a way for the robot to get around it. Normally, in this case the robot would stop and the full route-planning algorithm would be called on to find a new path to the robot's destination based on the new information about obstacles in the robot's path.

3. Localization Error

Every time the robot moves there is some slippage of the wheels that will cause the odometric encoders to incorrectly record the distance and direction the robot has traveled. Eventually, without some means of correction, the localization errors become so great that mapping and navigation become impossible. Methods of minimizing localization errors in mobile robots are the topic of much research [Ref. 20, 22, 27].

Figure 9 demonstrates the effects of robotic mapping with and without localization error correction methods while mapping a long, obstacle filled hallway. Figure 9(a) shows an evidence grid representation of the area to be mapped. Figure 9(b) shows a map developed by a frontier-based exploration system without the aid of any localization error minimization techniques. As the robot's coordinates become uncertain the sensor return data begins to "drift." Figure 9(c) shows the map which was developed using a continuous localization process that seeks to correct for dead reckoning errors that accumulate as the robot moves throughout the area to be explored. [Ref. 22]

Figure 9. Example of localization error and correction. Shown from top to bottom are the ground truth evidence grid, the map constructed without localization, and the map constructed with localization. [From Ref. 22]

43

D. NRL IMPLEMENTATION ON SINGLE NOMAD 200 ROBOT

The full implementation of the frontier-based exploration code (for both single and two robot systems) as developed at NRL consists of over 60 separate C and C++ routines that are then compiled into a single process. Many of these routines handle the various 'housekeeping' functions of any large, complicated program (i.e. display, input/output, etc.) and do not bear directly on this thesis. Relevant routines and their use will be discussed below and portions of the code from these routines will be reproduced in the appendices.

The latest version of the full code (as well as all previous versions) is available from Brian Yamauchi (yamauchi@aic.nrl.navy.mil) at NRL's Navy Center for Applied Research in Artificial Intelligence (NCARAI). The NPS modified version of the code is available from Xiaoping Yun (yun@ece.nps.navy.mil) at the NPS Department of Electrical and Computer Engineering (ECE).

1. System Overview

There are a few major processes in the NRL code that bear directly on single robot, frontier-based exploration. The three key parts of their single-robot system that are relevant here are: the use of laser-limited sonar (LLS) for sensor scanning at new frontiers, the exploration routine used for the detection of new frontiers and the subsequent movement to and scanning of those frontiers, and the integration of the new scan with the robot's current map.

44

2. Laser-Limited Sonar (LLS)

There has been much study of the characteristics of the simple type of Polaroid sonar units found on the NOMAD 200, the NOMAD SCOUT, and many other research and commercial robots [Ref. 28]. The low cost, low weight, and low power consumption of these types of sonars has made them very popular among robot builders and designers, however, they do have their drawbacks. As noted in Chapter III sonar returns in the real world environment tend to include much noise and many extraneous or false returns. Many of these questionable sonar returns are caused by a phenomenon known as specular reflection.

Specular reflection occurs when a sonar pulse hits a flat surface at an oblique angle and reflects away from the sensor instead of directly back to the sonar detector [Ref. 25]. When this happens there may be several different results depending upon the circumstances. If the sonar pulse reflected off of the oblique surface encounters a flat reflective surface soon after, then the detector may still get a return, but the first object that the sonar pulse struck will appear to be further away than it is in actuality. If the sonar pulse continues on to the sonar's maximum range without striking another object, then the nearby object may not be detected at all, but instead it will appear that there is a large open space surrounded by an unknown area.

In reality, the possibility of not detecting a nearby obstacle is not as great as it first appears. With many sonars onboard the robot firing from several different angles, there is usually not much problem with getting some measurable level of sonar from

45

nearby objects and detecting them even if there are facing obliquely to some sensors. However, the problem of detecting "phantom" obstacles and false open spaces is still a problem that needs to be considered. Figure 10 provides an example of a normal sonar return and two of the possible results of a specularly reflected return.

Figure 10(a) illustrates a normal sonar return with the sonar pulse represented by the ray emanating from the sonar, striking the nearby obstacle, and a measurable amount of energy returning to the detector. The head-on encounter of the sonar pulse with the flat face of the obstacle toward the sonar provides for a clear return path. In Figure 10(b), however, the oblique angle of the nearby obstacle reflects the sonar pulse away where it encounters the second obstacle. If a measurable amount of energy is returned from the second obstacle, then it may appear to the sensor that there is a "phantom" obstacle at the point shown in the figure. There may be a partial sonar return from the first obstacle as well, which could further confuse the sensor about the exact nature of nearby obstacles.

Figure 10(c) demonstrates what may happen if there is no second obstacle within the maximum range of the sonar sensor after the pulse has been reflected from the first nearby obstacle. The sensor would receive either a very weak or non-existent return from the nearby object. If no return is received then it will appear that a large open space exists in the area shown, when in fact there is really no information known about that area at all. Since this false open region would most likely be surrounded by unknown space it would also have the effect of creating false frontiers for the robot to explore.

Figure 10. Examples of a normal sonar return, a specularly reflected sonar return that creates a phantom obstacle, and a specularly reflected pulse that generates a false open space. (After Ref. [25])

In addition to the sonar sensors, the NOMAD 200 has a laser based range finding systems that does not suffer from these same type of specular errors. The researchers at NRL have taken advantage of this and created a technique known as laser-limited sonar (LLS). By using the readings from the laser rangefinder in combination with the readings from the sonar it is possible to eliminate most many false readings from walls and other large obstacles that cause the majority of specular reflections. If the laser returns a range to obstacle less than the sonar, then the evidence grid is updated as if the sonar had given the range indicated by the laser. [Ref. 23]

The laser cannot be used exclusively for mapping because the laser rangefinder currently available on the NOMAD 200 only operates in a two-dimensional plane, while the sonar senses obstacles within a three-dimensional cone radiating out from the robot. Objects above or below the plane of the laser will be missed by the laser, but detected by the sonar. Figure 11 provides an example of how this might happen. In this figure the laser plane is above the obstacle and thus the laser rangefinding system never detects it, but the sonar cone emanating from sonar sensor does intersect the obstacle. This is a case of using two different sensor types that compliment one another. A three-dimensional rangefinder would be an alternative that would combine the best aspects of both sensors, but presently these type of systems are too large, expensive, and power consuming to be commonly used on mobile robots. [Ref. 23]

Figure 11. Example of two-dimensional laser rangefinder failing to detect an obstacle that is within the detection cone of the sonar sensor. (After Ref. [25])

3. Frontier-Based Exploration Routine

The heart of the frontier-based exploration code is in the file *agent.cc*. It is in this operation that the frontier-detection, navigation, and exploration behaviors are described. This procedure also controls the laser-limited sonar scanning technique mentioned above and also takes care of integrating the newest scan with the current map via the method described below. The process begins by completing an initial sensor scan of the area upon startup and using the data obtained to construct an initial evidence grid map. After the initial map is created, a frontier detection subroutine is called to find and note nearby frontiers for further exploration. Once the frontier detection is complete the exploration

and navigation subroutines are called to choose the robot's next destination and get it there safely.

a.   *Frontier Detection Subroutine*

The method of frontier detection for this initial map and all subsequently generated maps involves using the edge detecting techniques described above on the most current evidence grid map that the robot has at that time. Mapped obstacles or edges separate frontier regions. The centroid (roughly the center of a non-symmetric region) of each frontier region is marked as the robot's target destination for exploring that region.

Figure 12 illustrates the different parts of the frontier detection process. Figure 12(a) demonstrates an evidence grid built by a robot in a hallway next to two open doors. Figure 12(b) shows the frontier edge segments detected in the evidence grid by the frontier detection process. Figure 12(c) shows the frontier regions that are greater than some threshold value (in this case roughly the size of the robot). The centroid of each of the frontier regions is marked with a crosshair and numeric label. The frontier regions labeled 0 and 1 represent the open doorways and the frontier region labeled 2 is the unexplored portion of the hallway. [Ref. 22]

b.   *Exploration Subroutine*

Once the frontier regions have been found on the most current evidence grid map, the robot must decide which frontier to explore next. The path planner in the exploration code uses a relatively simple depth-first search on the evidence grid. It starts

at the robot's current position and attempts to select the shortest obstacle-free path to

the centroid of the frontier region chosen as it next destination. [Ref 23]

(a)   *(b)*   *(c)*

Figure 12. Example of frontier detection. From left to right: the evidence grid constructed, the frontier edge segments, and the frontier regions with the centroid of each region labeled. (From Ref. [22])

c.   *Navigation Subroutine*

Once the exploration subroutine has chosen a frontier to explore it is the

goal of the navigation subroutine to get the robot to its intended destination in a safe and

efficient manner. Using the path chosen by the depth-first search and a variety of

reactive obstacle avoidance behaviors the navigation subroutine guides the robot. The

navigation method used in the exploration code is sufficient to allow the robot to steer

around small obstacles in order to get back on the pre-selected path. If the robot becomes blocked or for some other reason cannot make any progress toward its destination, then after a certain amount of time, that location will be added to the list of inaccessible frontiers that the robot cannot reach. At that point the robot will conduct another sensor scan, update its current evidence grid map, and attempt to navigate to the next accessible, unknown frontier as chosen by the exploration subroutine.

4. Integrating New Scan with Current Map

The map integration routine uses the method described in Chapter III for fusing the new sensor data onto the current map. The frontier-based exploration process uses a modified version of the log-odds Moravec code described in [Ref. 17]. In the NRL code, each cell of the evidence grid is assigned a value from -127 (definitely empty) to $+127$ (definitely full) for its occupancy value.

When the sensor fusion procedure described earlier is used to combine the current map data and the new scan data, the result will be a new map that reflects the effects of the most current sensor data. Similar data between the current map and the new scan will tend to reinforce one another driving well-mapped cells toward a floor value of -127 or a ceiling value of $+127$. Likewise, if the data in the new scan conflicts with the data in the current map, occupancy values of those cells will be driven toward smaller absolute values with zero representing a cell whose occupancy is completely unknown.

There is one important thing to note about coordinate systems used in the exploration process. When first initializing the robot, the user is asked to enter the

robot's X, Y coordinates and an initial orientation angle. For single robot exploration this is not necessarily required, as the robot could assume that its starting position and orientation are 0, 0 and 0 degrees and build a map around that point. However, for multiple robot exploration the robot knowing its starting position becomes very important as all maps sent to other robots are referenced to the shared global coordinate system. This will be discussed in more detail in the following chapter.

E.   NPS IMPLEMENTATION ON SINGLE NOMAD SCOUT ROBOT

In order to use the frontier-based exploration code developed for the NOMAD 200 on the NOMAD SCOUT many modifications to the original NRL code are necessary. Throughout the original code there are many obvious, as well as hidden, dependencies and assumptions based around the sensor suite and mobility base of the NOMAD 200. Most of the modifications can be broken down into one of two categories: those changes due to the differences in the movement commands between the NOMAD 200 and the NOMAD SCOUT and those changes due to differences in the available sensors on the two platforms.

1.   Mobility Modifications

As described in Chapter II, the NOMAD 200 has a three-wheel synchronous drive system with a turret that can rotate independently of the base, while the NOMAD SCOUT is a two-wheeled, two-degree of freedom differential drive robot which has no

53

turret. The NOMAD SCOUT was the first platform from Nomadic Technologies (manufacturer of all the NOMAD hardware and software products) that was built around such a mobility base. All their software prior to this had been designed for a three-wheel synchronous drive system with an independent turret. Early adopters of the NOMAD SCOUT, such as NPS, are using modified versions of the standard NDHE to interface with the new platform. Nomadic is currently developing a much more advanced version of the NDHE that will be more flexible in terms of working on multiple types of robots with varying mobility and sensor capabilities.

Fortunately, Nomadic has developed a set of macros that accept differential drive commands for the NOMAD SCOUT and transforms them into equivalent synchronous drive commands that the software understands. These modified movement commands convert the decoupled translation and steering commands used by the NOMAD 200 into differential drive values required for the NOMAD SCOUT. So when the software is controlling NOMAD SCOUT it is actually modified synchronous drive commands that are being sent to the robot. To eliminate any software conflicts due to the lack of a turret on the NOMAD SCOUT, the conversion macros send a null (zero) value in place of any turret rotation commands to the robot. [Ref. 10]

2.   Exploration and Navigation Modifications

The exploration and navigation modifications from the NOMAD 200 to the NOMAD SCOUT are much more extensive than the mobility modifications. The assumptions about the sensor suite of the robot designed into the original frontier-based

54

exploration code make for a large number of modifications to work on the NPS research platform.

a.   *Elimination of Laser-Limited Sonar Dependency*

In the original frontier-based exploration process the laser-limited sonar technique described above is used to scan whenever the NOMAD 200 reaches a new frontier. Since the laser rangefinding system is fixed in place on the NOMAD 200 turret, this involves rotating the turret a complete 360° while the base of the robot remains in place. There is also an option to use only the sonar sensors to scan at new frontiers. If the sonars are being used as the only sensor, there is another option to rotate the turret through the 22.5° arc that separates the sonar sensors and take sonar readings at intervals along that rotation.

In theory this gives the sonar sensors more opportunities to see obstacles from varying angles that might be less affected by specular reflection due to variations in the obstacle surface and what portion of the obstacle is struck by the sonar pulse. By modifying this sonar-only option to turn the entire robot instead of just the turret, it is possible to use this process on the NOMAD SCOUT in order to complete a sonar sweep upon reaching a new frontier. In addition, switching to the sonar-only option removes the dependency on a laser rangefinding system that the NOMAD SCOUT lacks.

b.   *Compensation for Inability to Timestamp Sonar and Pose Data*

Originally it had been thought that once the laser-limited sonar dependency had been removed, that it might be possible to have the NOMAD SCOUT collect sonar

range data while on the move and eliminate the need to stop at frontier boundaries in order to collect new sensor readings. Taking sonar readings while moving can present a problem because sensor readings are not instantaneous and the robot's position or the environment can change significantly between acquisition and processing of the sensor data, thus causing the collected data to be inaccurate [Ref. 12].

At the speeds both the NOMAD 200 and NOMAD SCOUT typically travel this does not cause any difficulty for the reactive obstacle avoidance routine, but it can cause problems for the accuracy of the mapping routine. On the NOMAD 200 it is possible to attach the robot's pose information and a timestamp to every sensor reading so that data taken while the robot is moving can be correctly interpreted. The NOMAD SCOUT lacks this capability. In order to ensure that the sensor readings and pose information were as closely matched as possible it was necessary to break the sonar sweep at new frontiers (as described above) into small, individual movements. After each movement the robot was halted, the sonar readings were taken, and the sweep was continued. This has the effect of slowing down the overall mapping effort and the repeated small movements tended to increase the localization error.

c.   *Specular Reflection Minimization and Side Effects*

In another one of the exploration code files (*grid.h*) it is possible to set the range at which sonar return data is considered "trustworthy." Because the NOMAD 200 has a laser rangefinder to confirm those sonar readings it is possible on the original code to leave this setting relatively high and disregard false readings. In the original code this

maximum sonar range is set at ten feet from the robot. Since the NOMAD 200 lacks a method of double-checking its sonar data this value is reduced to six feet. This also helps reduce errors due to specular reflection because it has been found that specular reflection errors are more prevalent at longer ranges [Ref. 28].

Unfortunately, there is an undesired side effect of reducing the "trustworthy" sonar range on the NOMAD SCOUT. With the decrease in range also comes a proportional decrease in the amount of new territory that is mapped whenever the robot reaches a new frontier. Mapping less area each time leads to an increase in the number of new frontiers to which the robot must travel in any given area to be explored compared to the number of frontiers with a longer sonar range. Increased travel leads to a faster buildup of localization errors when only dead reckoning from the robot's odometric encoders is used to determine the robot's position in the global coordinate system. Neither the original NOMAD 200 nor the current NOMAD SCOUT versions of the frontier-based exploration system incorporate any sort of localization error minimization process. Later work on the NOMAD 200 has included work with a continuous localization process [Ref. 20, 22] and it is hoped that this method or one like it may also be used on the NOMAD SCOUT in the future.

d. Reactive Obstacle Avoidance

While navigating to a new frontier, the reactive obstacle avoidance subroutine of the original exploration code uses the infrared sensors as well as the sonar sensors on the NOMAD 200 to detect nearby objects. The NOMAD SCOUT lacks

infrared sensors and it is necessary to remove any dependency on them and rely solely on the sonar sensors during the navigation and movement process.

It is also necessary to make some relatively minor modifications to routines that take into account the robot's size when determining if there is enough open space in a doorway or corridor for the robot to safely travel. The diameter of the NOMAD SCOUT is slightly less than that of the NOMAD 200 and thus it can move into a more constrained space.

V.   MULTIPLE ROBOT INTEGRATION

Even a very capable and very well equipped single reconnaissance robot will still be restricted in the amount of area that it will be able to cover in a given time by the limits of its mobility base and the range of its sensors. In addition, a single robot reconnaissance system is very vulnerable in that a single failure on the one platform can have catastrophic consequences on the ability of the system to perform its mission. By combining multiple robots together into a single integrated reconnaissance system, it is possible to have a greater area of coverage in a given time, quicker coverage of a given area, and continuous or overlapping coverage of high value target areas of interest. In addition, the use of multiple robots provides for a graceful degradation, rather than failure, of the system if individual robots fail to perform for some reason. In Chapter IV the mechanics of a possible single-robot exploration system were discussed. In this chapter the dynamics of using multiple numbers of such robots will be explored.

A.   CENTRALIZED VERSUS DISTRIBUTED CONTROL

There are two differing philosophies concerning command and control of multiple robot systems. The centralized approach advocates some sort of supervisor or controller process that receives inputs from the individual robots and provides information and commands back to them. The distributed approach calls for processing sensor data at the local level and individual robots making autonomous "decisions" based on that information. Between these two methods there is a wide range of combinations and

permutations depending on the intent of the system designers and how they choose to implement the system.

In the case of extremely centralized control there may be very little on-board "intelligence" on any of the individual robots and all commands of any sort (including motor and actuator commands) may have to come from the supervisor process. In this case the individual robots are little more than remote sensors on a mobility platform teleoperated by a central controller. The advantage of such a system is that each individual platform may be cheaper per unit. The main disadvantage is that highly centralized control leads to a single point of failure for the entire system and it will most likely also have a high bandwidth requirement if all raw sensor data and motor commands are required to be sent over the air [Ref. 29]. Another, less centralized, system may allow for centralized collection of processed sensor data from the individual robots which is then sent out to all the robots which then independently choose their own destinations for further exploration. Yet another system may call for the supervisor process to explicitly designate where individual robots will travel to and what tasks they will perform.

In a fully distributed system each individual robot might operate completely autonomously of the rest of the system. More autonomy on individual robots can lead to increased complexity and unit cost per platform, but it also allows elimination of the single point of failure problem that plagues highly centralized systems. Autonomous operation does not preclude cooperative effort between robots, but without a central supervisor it can complicate the problem of coordination. Lack of coordination in a

distributed system may lead to decreased efficiency and possibly even counterproductive behavior on the part of individual robots in respect to the goals of the system. This will be discussed in more detail in Chapter VI.

B. SENSOR FUSION

The evidence grid based mapping technique as described in Chapter III provides a good basis for the integration of sensor data from multiple, geographically separated robots. Sensor readings from different robots are fused in the same manner that sensor readings taken from a single robot at multiple locations are fused together. All that is required is that the sensor readings be referenced to a common global coordinate system in order for the sensor data from multiple locations to be properly correlated.

The details of how and where the sensor fusion is done will vary depending on the organization of the rest of the system. In a centralized system the coordinator/controller might receive all the individual robot sensor readings, fuse the data into a new global map, and redistribute that information throughout the system. A more robust distributed system might allow for each individual robot to receive all the other robots sensor readings (or at least those nearby), perform the sensor fusion locally, and "decide" for itself where to explore next based on some given criteria. Other implementations might allow for limited local processing of sensor data at the individual robot level with the resulting details transmitted to a remote higher level supervisory process.

C. COOPERATIVE EFFORT

In order to maximize the efficiency of a multiple robot system there has to be some degree of cooperative effort on the part of the individual robots. There are many possible degrees of cooperation as well as a multitude of methods and means of implementation. Both communication and coordination may be explicit or implicit with many varying combinations in-between.

1. Explicit Communication and Coordination

An explicit communication model allows for directed communications between each individual process and every other individual process as well as to and between any controlling or supervisory processes as well. While this model allows for a high degree of coordination and control, it can also become a communications nightmare very rapidly. The number of required links, L, for N separate processes in a fully interconnected system is given by:

$$L = \sum_{i=1}^{N-1} i \tag{1}$$

Another variation of this communication model is to use a few, or even just one, common broadcast channel(s) and then to attach some form of addressing to each message sent. Each individual robot or process then listens to the common channel(s) for messages addressed specifically to it, ignoring all others. This provides much the same functionality of the totally interconnected model with much less communications complexity.

Explicit coordination involves the passing of directions or orders, as compared to simply information, from a process outside the individual robot in order to influence or direct the robot's actions. It removes a degree of autonomy from the individual robots (which may no longer have a choice about their tasks and behaviors) and moves the decision-making ability to a higher level. Explicit coordination is usually associated with a hierarchical organization, but it can also be implemented on a peer to peer level where all the robots may be "equal," but at least on a temporary basis one robot may be able to direct the actions of another [Ref. 30].

Explicit communication and coordination is exemplified by the process of a parking lot attendant directing cars into and out of the parking lot. The attendant is explicitly communicating with each of the vehicle drivers through a series of hand and arm gestures (often accompanied by verbal expressions). The attendant is also providing explicit coordination amongst all the vehicles and has a direct line-of-sight communication channel with each driver.

2.   Implicit Communication and Coordination

Implicit communication calls for processes not to pass information directly to another process, but to still convey information in some manner to interested parties. This may involve some sort of display or broadcast on the sender's part, or the process or robot may have some sort of noticeable behavior that an outside observer can interpret [Ref. 31].

In general implicit communication has lower direct communication requirements from robot-to-robot or robot-to-supervisory process. However, there is a corresponding increase in the requirement for an individual robot to be able to interpret other robots' behaviors or displays and extract useable information them. This requirement may have a great effect on the unit cost per robot depending on the complexity of the information implicitly passed from robot-to-robot or robot-to-supervisory process.

Implicit coordination involves a single robot interpreting the actions and behaviors of other robots in the environment around it and taking individual action in accordance with the general goals of the system. This calls for a higher degree of autonomy on the part of the individual robot in order for it to know when to do the "right" thing at the "right" time. Implicit coordination is generally associated with a peer-to-peer organization where all robots are "equal," but it can also be implemented in a hierarchical structure with "lower" robots taking appropriate cues from the behavior of "supervisor" robots and vice-versa [Ref. 32].

Implicit communication and coordination is perhaps best exemplified by the process of an audience leaving a theater at the end of a movie or play. Generally, there is no overarching supervisor directing people into line and out of the theater and people do not explicitly announce their intentions to move into line to those around them. Instead, each person observers the actions of those around him/her and making a decision on when and where to move based on their actions and behaviors and following the general goal of leaving the theater.

D. NRL IMPLEMENTATION ON TWO NOMAD 200 ROBOTS

The interprocess communications routines used in the frontier-based exploration code were developed by Bill Adams (adams@aic.nrl.navy.mil) at NRL's NCARAI facility. The NPS modified versions of these routines are available from Xiaoping Yun (yun@ece.nps.navy.mil) of the NPS ECE Department as part of the NOMAD SCOUT modified frontier-based exploration code. Relevant routines and their use will be discussed below and portions of the code will be reproduced in the appendices.

1. System Overview

There are a couple of processes in the NRL code that bear directly on multiple robot, frontier-based exploration. The two key parts of their two-robot system that are relevant here are the communications process itself and the process of a robot integrating another robot's map with its own.

2. Communication Process

In the NRL code for two-robot, frontier-based exploration information about the world is shared, but each robot maintains its own map and makes its own decisions about where to navigate [Ref. 25]. There is no higher level supervisory process directing the individual robots or coordinating their actions. Normally this would be thought of as implicit communication and coordination. However, the system has to make use of an

explicit communication architecture to emulate the implicit communication process due to the limitations of the available networking protocols.

Even thought conceptually the process that is modeled is a peer-to-peer relationship the limitations of using existing TCP/IP networking tools preclude the system actually implementing this model. Instead, the original code simulates a peer-to-peer relationship using a standard client-server model. In the two-robot code the first robot is always designated as the server while the second robot is always the client.

Also, even though the model of the original code implies implicit communication, messages concerning new map availability are actually passed explicitly from robot to robot. It is also important to understand that in both the NOMAD 200 and the NOMAD SCOUT implementations that there is no controlling process actually running on the robots themselves other than low level motor controls in the robot's firmware. The controlling process of the robot is running on a remote UNIX workstation and all "communications" between robots is actually communication amongst the remote controlling processes. Also each controlling process is a client to the *Nserver* process. Figure 13 provides an illustration of how this is all connected together for a two-robot system.

As shown in Figure 13, all three processes, the *Nserver* and the two mobile robot processes, are running on the same UNIX workstation. In reality these three can all be on the same or separate workstations or any combination in-between as long as there is a shared memory location to which each robot process can write the map it will share with the other robot processes.

Figure 13. Illustration of the communication process used for the two-robot frontier-based exploration system.

During the exploration process whenever the individual robot completes a sensor

sweep at a new frontier, its controlling process writes the result of that local scan into the

shared memory location. This file (*local1.eg* or *local2.eg* depending on which robot

process that creates it) is an evidence grid representation of the most current local map of

the area surrounding the robot of the controlling process. In the original representation this file is not the full global map maintained by each robot process.

After the robot process has finished writing the updated local map to the shared memory location it sends a message to the other robot process that there is a new local map available. This is a case where an explicit communication is used to simulate what would normally be a broadcast if allowed for by the network protocols being used. Instead of the updated map data itself being sent, there is a directed message to the other robot process transmitted.

After the update message has been sent to the other process the controlling process checks to see if it has received its own message from the other process indicating that a new local map file is available. If there is one available it reads it from the shared memory location and proceeds to integrate that new remote local map into its current global map. Using this method writing new maps, sending messages to other robot processes, and checking for remote local maps to integrate is only done after a new sensor sweep has been completed at a new frontier.

One of the drawbacks to this method is that it is possible for a robot process to miss a remote map update from another robot process. This can happen if a robot process is directing its associated robot on a long traversal from one frontier to a new one to be explored. It is possible for the other robot process to explore a frontier, write an updated map, move its robot to a new frontier, explore the new frontier, and overwrite the previous update all before the other robot reaches a new frontier and completes its exploration, writing, and remote map reading routines.

The original information is lost because the local map file that is written to the shared memory location only covers the immediate area around the robot. Once the robot moves to a new frontier and a new local map file is written it is very unlikely that there will be any map area overlap between the new map and the previous one. This can lead to a robot process making decisions on where next to explore based on incomplete information as to where other robots have already explored.

3.   Integration of Foreign Maps

The integration of a foreign or remote local map uses the same methods described previously in Chapters III and IV for fusing new sensor data onto the current map. The sensor data from a remote map is fused with the robot process's global map in the same manner as if the process's associated robot had collected the same data itself. The important thing to note is that the remote local map must also have attached to it the global X and Y coordinates where the data was collected so that it may be registered correctly during sensor fusion with the global map.

E.   NPS IMPLEMENTATION ON FOUR NOMAD SCOUT ROBOTS

In order to use the communications routines developed for the two-robot NOMAD 200 frontier-based exploration code on a greater number of NOMAD SCOUT robots a few modifications to the original NRL are necessary. None of the changes are actually specific to the NOMAD SCOUT so the modifications made here will work just

as well on an increased population of NOMAD 200 robots. It is hoped that other researchers with larger numbers of NOMAD 200 robots will be able to make use of the modified code developed at NPS. The modifications can be broken down into two parts: extending the client-server architecture to mange more than two robot processes and transmitting the global map vice the local map from the server robot process.

1. Extended Client – Server Model

There are a variety of possible different approaches to extending the original code's client-server model for two robots to a client-server model for greater than two robots. In order to maintain the fully interconnected architecture of the original NRL code it would be necessary for individual robot processes to function as both clients and servers depending on the circumstances. An example of how this might work for four robot processes is shown in Figure 14. A process being both server and client simultaneously is not without precedent as all the robot processes are also clients to the *Nserver* and the first robot process is also a server to the second robot process in the original model.

However, as seen in Figure 14, the number of interconnections increases rapidly as the robot process population grows. It becomes apparent that this is an unnecessarily complicated and unwieldy implementation for a large number of robot processes and an alternative method is used that while not fully interconnected between robot processes, still provides suitable connectivity for the transmission of remote map file information.

Figure 14. Illustration of a fully interconnected communications architecture for four robot processes using a client-server model.

Instead a single robot process is used as a server and all the other robot processes are clients to that single server process. This model is shown in Figure 15. As in the original NRL code the first robot process acts as the server. Also, it should be noted that all the robot processes remain clients of the *Nserver* program. This approach eliminates

the full interconnection of the original NRL model, but greatly simplifies the modification

of the code to work for larger numbers of robots. However, without the direct connection

from client process to client process there is a need for another way to transfer remote

local map information between client robot processes. The solution to this problem is

discussed in the next section.

Figure 15. Illustration of the communications architecture implemented for four robot processes using a client-server model and retaining the single robot process as server.

In this model each of the client processes is in effect part of a two-robot system,

itself and the server process. Only the server process "sees" all the individual client robot

processes. However, there is still a good deal of robustness to the system. If one of the

client process robots fails the rest of the system will continue to function in the same

manner with the smaller robot population. If the server process fails there will be no

more sharing of map data, but each robot process will continue to explore individually

until there are no more frontiers remaining.

2. Transmission of Global vice Local Maps from Server

In the original NRL code whenever a robot process writes a map file to the shared memory location it is a local map file of just the immediate area surround the associated robot of that process and showing sensor data from the latest sensor sweep only. It is possible to modify the code so that instead of just writing the local map data, that the global map maintained by that process is written to the map file. This global map file incorporates all the separate local sensor sweeps that that process has made up to that time as well as any remote map files that it has fused into its global map.

Modifying the code in this manner makes it possible for each robot to write the entire evidence grid representation of its global map to the shared memory location (*globalx.eg* where *x* is the number of the robot process). In this manner the server process incorporates the individual global maps of all the individual client processes into its own global map. When the client processes read the *global1.eg* remote map file written by the server process they then indirectly incorporate all the results of the mapping done by the other client processes.

This also solves the problem of individual sensor sweeps being "lost" due to a short move, explore, write cycle causing the file to be overwritten. In the case of writing the entire global map the new global map will incorporate the previous sensor sweeps as well. However, having all the separate robots write their individual global map files has an unexpected drawback as well.

When only the local sensor sweep information is propagated from robot process to robot process then "bad" mapping data from one robot (due to sensor errors, location errors, etc.) may be overwritten when another robot happens to scan the same area. Also in this manner temporary obstacles (people, other robots, etc.) are steadily eliminated or their positions updated on the global map that each robot process maintains. By having each robot send its entire global map an unwanted feedback loop for the reinforcement of "bad" data can occur.

If a client robot scans an area that happened to have a temporary obstacle or is very "noisy" due to specular reflection off of objects in that area, the results of that local scan will be incorporated into its own individual global map. Now when that global map is read by the server process it will fuse that data with its global map and write its updated global map to the shared memory location for all the client processes to read. After they read it and update their global maps, the next time they write their global maps for the server process they will also show the same "noisy" or temporary obstacle sensor data which will reinforce the previous data on the server process global map. The server process will in turn write this updated global map for the client processes to read and the feedback of incorrect or out-of-date sensor data will continue. This is not a desired situation.

In the final version of the NPS modified code only the server process writes its global map for the client processes to read. The client processes write out only the results of their local sensor sweeps for the server process to read. In this implementation temporary obstacle data or "noisy" sensor data will be propagated through the system

once when the server robot incorporates it into its global map, but it will not be reinforced by having that same sensor data sent back to it from the client processes. The tradeoff to this approach is that once again the server process may miss a client process local map if the server process is busy controlling its robot during a long movement between frontiers.

VI.   RESULTS

Despite equipment delivery delays and the need for extensive software development, substantial initial testing of the NOMAD SCOUT multiple robot frontier-based exploration system at NPS was possible in the limited time available for research. Besides the results presented here, the major product of this research was a demonstrable robotic exploration system that will serve as a testbed for future projects involving both single and multiple robots. Presented here are the preliminary findings to date.

A.   SINGLE ROBOT MAPPING EFFORT

Single robot mapping of a given area provided a baseline against which multiple robot mapping efforts were compared as well as ensured that the basic frontier-based exploration code and evidence grid map making routine functioned properly in conjunction with the NOMAD SCOUT robot. Early tests were also used to determine the best combination of map grid resolution, "trustworthy" sonar range, and given area to be explored that would yield optimal results for a single robot. The best combinations of these variables were then used for each individual robot in multiple robot mapping experiments.

1.   Single Robot Test Conditions

The test area for all single and multiple robot trials was an approximately 37 by 37 foot research room with two large test benches that defined three major corridors in the

space. An additional test bench along one wall further constricted one of the corridors. Figure 16 is a simple illustration of the area with annotations for the various starting positions used in the trials. The center of the room was defined to be the origin of the coordinate system used in the map produced during the robot mapping trials. This origin is marked as position zero in the illustration below. The various corridors are referred to as top, middle, and bottom as labeled in the sketch of the room. Note the windows stretching along one wall, the metal cabinets, and the large number of doors. These were geographical features that proved especially challenging during efforts to map the area due to specular reflection effects.

Shown in Figures 17, 18, and 19 are a series of pictures taken of the test area in order to provide the reader with a better perspective of the environment. Note the large open spaces under the test benches. Because the lower portions of these benches were so close to the ground the sonar sensors often failed to detect them. It was necessary to fill in some of the space under the benches in order to enhance their sonar image before any worthwhile results were possible.

Figure 17 shows the top corridor of the test environment. The metal desks and windows in this area proved particularly difficult to accurately map. Figure 18 shows the middle corridor of the test environment. This corridor runs down the center of the test environment and the midpoint of this corridor served as the origin of the coordinate system used in all the mapping trials. Again, the windows at the end of this corridor caused difficulties in the mapping trials. Figure 19 shows the bottom corridor of the test

environment. The smooth-surfaced doors and the metal cabinet in this corridor were the major sources of mapping errors.

Figure 16. Simple illustration of test environment for single and multiple robot trials showing starting positions and significant geographical features.

Figure 17. Top corridor of test environment.

Figure 18. Middle corridor of test environment.

Figure 19. Bottom corridor of test environment.

2. Experimental Variables

After the basic robotic exploration and mapping system was functional it was decided to concentrate on examining a few easily-manipulated variables in order to attempt to optimize the system for the given test area.

a. Given Area to be Mapped

The first thought was to minimize the area the robot would be expected to map. This variable is set in the file *grid.h* and sets the size of the room the robot will map. Minimizing this would seem to ensure the finest detail possible for the given evidence grid resolution (as described below). It proved not to be practical to set it exactly to the actual size of the test area. In a perfect world the robot could have been

instructed to map a 37 by 37 foot room and all would be well. However, as odometry errors began to accumulate during a test run, the robot became confused near the edges of the room when its now-inaccurate odometric encoders indicated that it was outside the boundaries of the expected area.

In addition, specular reflection errors off of objects near the boundaries (especially the windows) caused false sonar returns from outside the boundaries set for the room. This caused additional errors. To alleviate these difficulties a 3.5 foot safety margin was added on each side of the room boundaries, resulting in a 44 by 44 foot area that the robot expected to map. This reduced the overall resolution slightly, but resulted in more consistently successful mapping efforts.

b.   Evidence Grid Resolution

Also in the file *grid.h* the evidence grid resolution is set. In order for the evidence grid method to correctly fuse two different grids the grids must be symmetrical and a power of two. Varying resolutions from 64 by 64 cells to 512 by 512 cells were tested.

The initial testing with a setting of 512 by 512 cells resulted in very noisy sonar data and many small frontiers. These small frontiers were often found to be grouped around one large object, especially one with many projections such as a chair or table. It was hoped that setting a coarser resolution would result in quicker mapping of large areas and less noise from the arms or legs of chairs and tables. It soon became

evident that using a very coarse resolution, such as 64 by 64 cells, did result in a shorter mapping trial, but not for the desired reasons.

With the room size mentioned above of 44 by 44 feet and using 64 by 64 cells in the evidence grid, each cell was about 68 in^2 or 8.25 inches on a side. This is a rather large size for a cell compared to the size of objects in the test environment. As expected, the noisy sonar returns were blurred into fewer cells, but the unfortunate side effect was that now large cells that were only partially filled were marked as completely filled, whereas with finer detail these areas would have been resolved into open space. With the coarse detail setting the robot soon marked all possible paths as blocked by obstacles when in fact there was still many open paths for it to travel. This is illustrated in Figure 20. Here the resolution was set at 64 by 64 cells and the robot was started at position zero in the center of the room. Even though the corridor was open, noisy returns were still blurred together to the point where the robot determined that is was completely blocked.

Numerous trial-and-error investigations led to the choice of 256 by 256 cells for the grid resolution in conjunction with the "trustworthy" sonar range discussed below. Again using the room size of 44 by 44 feet, but now with 65536 cells, each cell in the evidence grid was approximately 4.25 in^2 or less than 2.1 inches on a side. This was the best compromise found between reducing noisy data and having fine enough detail to navigate the robot and map properly. It is important to note that these results were specific to the environment the mapping tests were conducted in and will most likely be quite different in dissimilar circumstances.

Figure 20. Illustration of robot exploring corridor using coarse discrimination (64 by 64 cells). Large individual cell size causes false determination that ends of corridor are blocked.

Figure 20 continued. Illustration of robot exploring corridor using coarse discrimination (64 by 64 cells). Large individual cell size causes false determination that ends of corridor are blocked.

c. "Trustworthy" Sonar Range

The final option that was manipulated in the *grid.h* file was the *MAX_SONAR_RANGE* variable. This variable sets the "trusted" range for sonar sensor readings. Readings indicating distances further away than this setting will be disregarded for the purposes of map building and exploration. As was mentioned earlier, setting this to a lower value than the 10 foot range used with the NOMAD 200 seemed to be the best way to reduce the problem of specular reflection. Also, as mentioned previously, there was a penalty in setting this too low in the increased amount of travel the robot would be required to do and the subsequent increase in localization error.

After many trials it was found that a six foot range was adequate to reduce many (but not all) specular reflection effects and did not seem to compromise the robot's localization capability to any great degree. However, if an additional localization method is added to the NOMAD SCOUT platform in the future it is recommended that this range be further reduced in order to further mitigate specular reflection problems. Figure 21 shows a sequence of maps created during a NOMAD SCOUT mapping sequence with the *MAX_SONAR_RANGE* set to 10 feet and a grid resolution of 256 by 256 cells. The robot was started at position zero in the center of the test area. Note the numerous and extensive specular reflection effects from the areas near the benches and windows. These false returns created numerous small, false frontiers that the robot attempted to explore, but could not reach.

Figure 21. Illustration of false sonar returns and subsequent poor mapping results due to extensive specular reflection when using 10 foot sonar range.

Figure 21 continued. Illustration of false sonar returns and subsequent poor mapping results due to extensive specular reflection when using 10 foot sonar range.

3.   Trial Runs and Results

One thing is immediately noticeable from the many trial runs conducted with one robot in the initial research: as currently implemented no single robot alone will be able to map the entire test area. On average, after 20-25 minutes of travel the odometry errors

become so large that further mapping efforts are actually counterproductive. Continued mapping at that point, with localization so badly compromised, will most likely begin to overwrite previously accurate areas of the evidence grid map with inaccurate data. This was seen many times in longer trials. In the current implementation there is not enough time before odometry error becomes fatal to the exploration and mapping effort for the robot to cover the entire space. Thus the need for multiple robot exploration and mapping is evident.

Figure 22 illustrates a typical trial run with the "standard" settings of a 256 by 256 cell evidence grid resolution and a maximum trusted sonar range of six feet. This run was started from position zero in the center of the test area. Mapping efforts continued well for about the first 15 minutes. After that time, localization errors began to interfere with the robot's ability to navigate to new frontiers. Localization continued to get steadily worse, especially rotational tracking. By the 21st minute of the experiment the robot was actually travelling in the opposite direction than it indicated that it was moving. This seemed to be a common trend among many of the trials. The small movements that the robot makes during sonar sensor sweeps at new frontiers as well as the many small turning motions the robot makes as it travels seem to affect the rotational localization much more quickly and much more detrimentally than the translational localization.

Figure 22. Illustration of fatal localization error beginning about 15 minutes into the mapping and exploration phase.

15 Minutes

17 Minutes

21 Minutes

24 Minutes
Physical robot was travelling in
opposite direction than indicated

Figure 22 continued. Illustration of fatal localization error beginning about 15 minutes into the mapping and exploration phase.

However, other trials showed that localization errors could also cancel each other

out and allow longer periods of mapping. These maps will be distorted compared to the

actual "ground truth" of the area mapped, but they will still have recognizable, albeit

distorted, geographical features such as corridors, corners, etc. Figure 23 is an example of

such a trial. This run was conducted under the standard conditions with the robot

initially started at position one. Between the nine and 19 minute point in the trial the

robot was stuck in a small area between the two benches trying to explore many small,

inaccessible frontiers. By the time it "broke free" its odometry was obviously distorted,

but it was possible to still recognize map features produced for another 12-14 minutes.

Figure 23. Illustration of robot getting temporarily trapped in a small area, breaking free, and then continuing to produce a recognizable, although distorted, map for several minutes longer.

Figure 23 continued. Illustration of robot getting temporarily trapped in a small area, breaking free, and then continuing to produce a recognizable, although distorted, map for several minutes longer.

All single robot trials continued to point toward the need for multiple robots acting simultaneously in order to map the test area. Under the current implementation a single robot cannot map the area before localization errors render it incapable of further exploration and mapping.

B.   MULTIPLE ROBOT MAPPING EFFORT

Multiple robot mapping efforts provide a number of mixed results. In some circumstances the use of multiple robots dramatically decreases the amount of time required to map a given area compared to a single robot mapping the same area. In fact in some cases multiple robots were able to map an area that the single robot could not complete due to buildup in localization errors. However, under other circumstances multiple robot mapping can be less efficient than expected and actually worse than single robot efforts. There also appear to be some issues with the effects of controlling many robots simultaneously on network performance and reliability.

1.   Multiple Robot Test Conditions

The test area for all multiple robot trials was the same as for the single robot trials, the 37 by 37 foot research room described previously. All robot processes, as well as the *Nserver* program, were run on the same Sparc 20 workstation used for the single robot trial and the robot processes controlled their respective robots via the same wireless Ethernet connection. Using the same workstation to run all the robot processes simplified the sharing of map data between the client and server processes, but did lead to an overall slowdown in the speed at which the individual processes ran as more robot processes were added. For all the multiple robot trials each individual robot was similarly configured with an evidence grid resolution of 256 by 256 cells and a trusted sonar range of six feet.

2.   Trial Runs and Results

One major finding from the multiple robot trials was that there was not quite the consistency or quality of performance improvement that had been expected. In a perfect implementation if a single robot can map X area in Y time, then N robots should be able to map X area in Y/N time (or conversely map $N*X$ area in Y time). While such extreme levels of performance improvement were not expected at this point in the research, it was expected that there would be somewhat more improvement and more consistency in improvement than was seen. This will be discussed further below.

3.   Beneficial Effects

Some multiple robot trial runs did show significant improvement in mapping efforts over single robot trials. This was especially true when the robots started in widely varying geographical positions in the test environment with well-known initial starting coordinates. In these cases each robot mapped its local area in the same manner as in the single robot trials and the server robot process consolidated the map data. Figure 24 illustrates this process for an average two-robot trial.

In the trial shown in Figure 24 one robot was started at position zero and the other robot was started at position five. With the two robots separated by an obstacle (in this case one of the benches) they explored their general area without interference from each other. One important thing to note about this trial is that the robot in the middle corridor suffered networking problems after approximately 20 minutes. It stopped

mapping, but the robot in the top corridor continued mapping. This illustrates the benefits of a distributed system without reliance on a central controller to operate.

As the still functioning robot continued to map the top corridor localization errors soon began to degrade its navigation capabilities after about 23 minutes. At the 30 minute point in the trial further mapping efforts are futile. Note the specular reflection effects on the robot in the middle corridor near the windows and along the benches in the middle corridor.

Figure 25 illustrates a representative three-robot trial with the robots starting in widely separated positions. In this case the robot were started at positions two, eight, and nine. Again, each robot began to explore its local area without interference and the server robot process collected the local sensor data from each client robot process, fused the data, and distributed a new global map to all the robots in the system.

For the 20 minutes that this experiment ran localization for each individual robot was maintained relatively well. The combination of the three robots managed to explore and accurately map more of the test area than a single robot would have been able to in the same amount of time. More importantly, even if a single robot could manage to navigate through the same amount of area that the three robots covered, its mapping accuracy would be much worse than the three-robot system. The longer time required for a single robot to cover the same area as three robots would lead to many more localization errors on the single robot compared to those of any individual robot in the three-robot system.

Figure 24. Illustration of a two-robot exploration trial with the robots starting in widely different positions.

Figure 24 continued. Illustration of a two-robot exploration trial with the robots starting in widely different positions.

Figure 25. Illustration of a three-robot exploration trial with the robots starting in widely different positions.

99

Figure 25 continued. Illustration of a three-robot exploration trial with the robots starting in widely different positions.

4. Counterproductive Effects

Not all the results of using multiple robots were beneficial in terms of mapping

accuracy and efficiency. There were several sets of circumstances that often led to

multiple robot trials being less efficient than it would be assumed they would be and in

100

some cases even less efficient than a single robot system. The most common cause of multiple robot inefficiencies was near proximity of one robot to another robot during the exploration and mapping process.

a. "Follow The Leader" Behavior

Two related multiple robot exploration problems came to be known as the "Follow The Leader" and "Dancing Robots" behaviors. Both of these behaviors happen when two or more robots are near (within the trusted sonar range) one another. The "Follow The Leader" behavior can be described as one robot appearing to follow another robot through the test environment and apparently mapping the same area that the leader robot has just mapped. Numerous real-life and simulation trials have revealed two predominant causative factors for this behavior.

The primary cause seems to be the time delay in map data being passed from one robot to another and when that data is processed during the exploration routine. When two robots are near one another they will usually see the same frontiers nearby. Commonly one robot will proceed to a nearby frontier and the other robot will proceed to another nearby frontier that is slightly closer to it. However, if the second robot finishes its exploration of the first frontier and still has not received the map data from the first robot's exploration there is a good chance that it will travel to the same frontier that the first robot just explored.

The second robot will probably not receive and process the first robot's map data until after the second robot has already mapped the same area that the first

robot just left. By then the first robot has moved on to a nearby frontier, which is now also the next frontier that the second robot will attempt to explore. This process can continue with the second robot always one set of map data behind the first robot and following it all over the test area.

The other common cause of this behavior is that the second robot senses the first robot that is nearby as an obstacle in the environment with unexplored frontiers around it. While the second robot heads for the first robot's position the second robot moves away to explore a nearby frontier. Once the second robot reaches the first (leader) robot's previous position it makes a sensor sweep, again notes the nearby first robot as an obstacle with new frontiers to explore around it, and again the "Follow The Leader" process continues. In the best case this type of behavior merely reduces the effectiveness of the system by one robot (the following robot). In the worse case, instead of the "Follow The Leader" behavior, the "Dancing Robots" behavior occurs and both robots are rendered ineffective.

b.   *"Dancing Robots" Behavior*

The "Dancing Robots" behavior can be described as two or more robots circling around or moving back and forth near one another for extended period of time and remaining in a relatively small area of the test environment. One variant on this behavior is vaguely reminiscent of the "do-si-do" movement, typically seen in square dancing, where two robots will swap places as if they are swinging each other around. As

interesting as this behavior is to observe, it does not aid in the exploration and mapping process.

This behavior is also caused by a robot sensing another robot as an obstacle with new frontiers around it to be explored. However, in this case both robots sense one another instead of just a follower sensing a leader. Whenever one of them moves to explore the frontiers around the other, the other does the same and thus the robots "dance" around one another. This process can continue indefinitely until one robot is stopped or another nearby frontier that is not caused by a mobile robot is chosen for exploration. Besides keeping two robots from exploring the rest of the area, the constant "dancing" motions have an extremely detrimental effect on localization.

Figure 26 is an example of this inadvertently happening in a three-robot trial. The robots were initially started at positions one, six and seven. Exploration continued normally for the first few minutes. After about 5-7 minutes the robots in the two lower corridors came close enough together to sense one another. At that point many small frontiers were generated by each robot during its exploration process as the other robot moved through the area while the sonar sensor sweep was taking place. The robots began to "dance" around one another for the next 6-7 minutes until one of them was halted. At that point the other robot was able to "break free" because no new frontiers were being generated by a moving robot. However, by this point the robot's localization has been compromised due to the effects of small movements around the other robot.

Figure 26. Illustration of "Dancing Robots" behavior during a three-robot trial.

104

Figure 26 continued. Illustration of "Dancing Robots" behavior during a three-robot trial.

Figure 27 is a trial done with all the robots started in a very near proximity to one another. The initial starting points were positions one, two, and three. *Nserver* can be used to display the robots relative positions in the test environment based on the encoder data sent from the robot back to its respective controlling process. This option was used to track the robots' positions in the test area. The robots are labeled on the figure for ease of reference.

At the start the first robot is slightly farther away from the other robots and does not see any frontiers generated by detecting the other robots as obstacle. In comparison, the second and third robots are much closer together and generate many small frontiers as they detect each other nearby. At first it appeared that the second robot might just follow the third robot, but the third robot detected no frontiers closer than those around the second robot and thus began the "dance."

Robots Initial Positions

2 Minutes

Robots 2 and 3 see
many small frontiers
around each other

2 Minutes

3 Minutes

Figure 27. "Dancing Robot" behavior from robots in near proximity.

1

2

3

5 Minutes

5 Minutes

1

3

2

8 Minutes

Figure 27 continued. "Dancing Robot" behavior from robots in near proximity.

c.   *Propagation of "Bad" Data*

Throughout all the multiple robot trials it was evident that a single robot could enter "bad" data into the system. Localization reliability varied greatly from robot to robot and trial run to trial run depending on many variable such as the area being mapped, wheel slippage, etc. As seen in many of the multiple robot results shown here a

107

map made by multiple robots can be very accurate in some areas and very inaccurate in others depending on the varying quality of mapping data sent from the individual robots.

d.   Network Reliability Problems

Throughout the trial runs there were unexplained network problems that seemed to increase in severity as the number of robots used was increased. Despite many attempts to track and mitigate the problem they continued to greatly detract from the ability to operate three or more robots for extended periods of time. For three robots 20 minutes was normally the longest time the robots would operate before packet errors caused termination of the experiment. This problem remains under investigation.

C.   LESSONS LEARNED

There were several immediate lessons learned from this initial research. The first of these was that rotational odometry errors are much more detrimental to mapping efforts than translational errors. While translational errors do affect the quality of the map the robot is still able to navigate. Rotational odometry errors quickly increase to the point that that the robot is completely confused as to which direction it is facing and further navigation becomes impossible. However, any rotational localization scheme (such as wall or corner detection) will probably have a beneficial side effect of aiding translational localization as well.

The second lesson is that the better a robot can explore and map on an individual basis the better it will function as part of a multiple robot exploration and mapping

system. This is basically common sense. Continued improvements in single robot

mapping will also improve multiple robot mapping.

The third lesson is that the network reliability issue needs further investigation. It

needs to be determined whether the robot trials were causing the problem or if the cause

was from an outside source. As mentioned above the local network administrators are

currently investigating this problem.

VII. RECOMMENDATIONS FOR FURTHER STUDY

Due to the extensive amount of software modifications necessary and the constrained equipment availability there were a number of areas of research which promised to be very interesting, but which there was not time to pursue. It is hoped that future students will take up the task of continuing some of the possible avenues mentioned here now that the initial work has been done in order to provide a testbed system for research. These future research possibilities can be broken up into two main categories: those that would involve mainly software modifications only and those that would involve hardware additions or modifications in addition to software changes.

A. SOFTWARE CHANGES ONLY

Many possible research areas would require only software changes to the existing code and require no additional hardware. Also, there is still ample opportunity for optimization of the existing frontier-based exploration routines as currently implemented.

1. Centralized Map Building Process

There are many possible methods to centralize the map building process and possibly reduce or eliminate some of the counterproductive behavior seen in the initial trials while still allowing the individual robot processes to function with relative autonomy. One possibility is to implement a sort of "Blackboard" to which the individual robot processes would write, or send, map information.

The Blackboard would be a separate process or perhaps a "virtual" robot that would only accept local remote maps from all the robots and control no individual robot of its own. It would use the local maps sent to it to build a global map, which could then be sent back to the individual robot processes. It would take the place of the of the first robot process in the current implementation, acting as a server with all the individual robot processes as clients to it. How this might look for a system with four individual robots is illustrated in Figure 28.

Blackboard Process To and From Individual Robot Processes

Blackboard Process

Robot Process 1   Robot Process 2   Robot Process 2   Robot Process 2

Figure 28. Illustration of a Blackboard-type process interacting with four individual robot processes.

This procedure could have a beneficial effect on the map building process in a number of different ways. By using only local scans to build the global map many of the problems of temporary obstacles gaining persistence and the unwanted reinforcement of "noisy" data are mitigated if not eliminated. Also, since the Blackboard process would

112

not be controlling a robot directly it can scan the shared memory location or "listen" for new local map messages constantly. This eliminates entirely the problem of missing local updates and losing map information from the individual robots. The Blackboard process would also provide a central point from which a user or operator could monitor the robotic mapping efforts of both the system and individual robots.

The Blackboard process could also be used to track the location of individual robots. Using this information the global map could be marked to either show the area physically covered by a robot as obstacle free or already explored. When this global map is sent back to the individual robots this information could be used to eliminate the problem of robots sensing each other as obstacles and the "dancing robot" behavior that follows. Having the Blackboard process create the global map also makes for a convenient location for any new robots joining the system to find out the most current map and avoid duplication of earlier efforts. More of the challenges of managing a dynamic robot population are discussed below.

The basics of the Blackboard process should actually require very little coding. Most of the functionality of taking in local maps and creating a global map is already done in the current implementation by the first robot process when acting as a server. Also, the client robot processes already write their local maps to and read the global map from the same shared memory location. For a basic Blackboard process simply removing the code that controls an individual robot from the original robot code and having the process constantly scan for and process local maps from the client processes would be sufficient.

2. Centrally Coordinated Effort

Another aspect that is worth investigating is removing some autonomy from the individual robot processes and creating some sort of centralized control or supervisory function. In this case the supervisory process would be able to direct or at least influence the individual robots' actions. This control could be constant, thus removing all autonomy from the individual robot processes, or on an as needed or exceptional basis, otherwise allowing the robots to act independently.

This control process would most likely act in conjunction with some sort of robot tracking process, perhaps one much like the Blackboard process described above. Besides eliminating the "Dancing Robots" and "Follow The Leader" behaviors it would also provide a single point from which an outside user or operator could direct the actions of an individual or groups of robots as well as see the results of the robotic mapping efforts. This is an important option in a deployable reconnaissance system.

This modification of the original implementation would require more extensive changes than just adding a simple tracking system. Besides the creating the supervisory process itself, it would also be necessary to make extensive modifications to the individual robot processes to have them accept commands from an outside source.

3. Dynamic Robot Population

The current implementation does not allow for an easy or simple method of joining additional robots to the system after initialization or for a way for a robot to

114

gracefully leave the system (say to go on to another task or report for maintenance). What is needed is some way to easily manage a dynamic or changing robot population.

Whenever a robot enters the system it needs some sort of unique identifier within the system so that other robots and any supervisory or other processes that exist have a way to identify or track the new robot. This identifier also serves to identify any local maps made by the robot. Under the current implementation the robot identifier is assigned by the *Nserver* in the order that the robots are created in the program. The user then assigns the individual robot processes a number corresponding to the one given by the *Nserver* program. For a dynamic robot population a dynamic method of allocating unique identifiers to robots is required.

One possibility would be a to use a simple Boolean array stored in the shared memory location used by all the robots. The size of the array would be the maximum number of robots that the system could manage. The array would be initialized to all zeros representing no robots in the system. As robots enter the system they would first scan the array until they found the first zero position. The robot would set the zero to a one and the numeric position in the array of that zero would become the robot's unique identifier.

Likewise, a robot leaving the system would reset its identifier position in the array to a zero, thus opening up that identifier for a new robot joining the system to use. If the system is filled with all the robots it can use or manage new robots would find the array filled with ones and would act appropriately depending on the system design, either waiting until an opening is available, moving on, or taking some other action altogether.

How this might function for a system with a maximum capacity of four robots is illustrated in Figure 29.

System With Four Robot Maximum Capacity					

Robot Identifier Array At Initialization

1	2	3	4
0	0	0	0

First Robot Enters System and Takes First Identifier
Becoming Robot One

1	2	3	4
1	0	0	0

•
•
•

Process Continues Until Four Robots Are In System
-Any New Robots Attempting To Enter At This Time
Will Find The Array Full And Be Turned Away

1	2	3	4
1	1	1	1

Robot Three Leaves System And Opens Identifier
Position

1	2	3	4
1	1	0	1

New Robot Enters System, Finds Open Position,
And Becomes Robot Three

1	2	3	4
1	1	1	1

Figure 29. Illustration of the use of a Robot Identifier Array in the managing of a system with a four robot limit.

Furthermore, this array could hold other information rather than just the individual robot identifier data. In conjunction with the sort of supervisory process described above it might also be useful to have additional information such as sensor types and ranges available on the robot, mobility platform capabilities and limitations, and other types of data that could aid the central controlling process in managing the robotic resources available to it.

4. Communications Networking Model

The communications model in the current implementation is based on a point-to-point system in which an individual robot process explicitly communicates with only one other robot process at a time across a hardwired network connection. To better simulate a deployable system, where all links are wireless, a broadcast communications model should be used. One possibility would be to simulate the effects of range loss by including the individual robot's coordinates in the global map as an attachment to any message. The receiving robot could compare the sending robot's location to its own and decide whether or not to "accept" the message based on the distance between the two robots.

Another area worth further research is the general appropriateness of TCP as a communications protocol for mobile robot systems. Mobile robots in a real world wireless environment do not fit well with the design behind of TCP and its orientation around continuous streams of data. Mobile robots require a more message-based communications design. There are a number of other possible networking models and protocols other than the TCP/IP model currently used. Some work in this area has already been done, focusing in on the use of the User Datagram Protocol. [Ref. 33]

5. Improved Localization Method

As has already been mentioned the current implementation has no method of localization beyond simple dead reckoning using the robot's on-board odometric systems.

As shown in Chapter VI, this quickly proves insufficient as a means to accurately track the robot's location and map making efforts suffer accordingly. There has already been much work done with localization routines on a NOMAD 200 robot at NRL, both alone and in conjunction with frontier-based exploration [Ref. 20, 22]. It is hoped that their methods might be adapted to work with the NOMAD SCOUT robot as well.

Other research at NPS has concentrated on determining a robot's location in the real world through interpretation of its surroundings [Ref. 8, 9]. This work also shows promise of forming the basis of a general localization routine for any number or type of mobile robots. Other efforts have attempted to have the robot match its current surroundings to an evidence grid built by the robot and correct any rotational or translational errors that may have developed [Ref. 34].

Another possibility involves the outside use of some sort of supervisory process such as described above. This process could monitor a robot's self-reported location, the robot's reported surrounding, as well as any sensor reports from other robot's in the area. Using this information the supervisory process might track each of the robots in the system and send correction information to them as their dead reckoning systems begin to drift. Still another possibility is a group of robots forming a local reference system without the use of any central process. Some work has already been done in this area [Ref. 35].

6. Managing a Heterogeneous Robot Population

The multiple robot frontier-based exploration system has been implemented separately on groups of NOMAD 200 and NOMAD SCOUT robots. So far there has been no work done on integrating a heterogeneous population of robots in such a system. There are important questions concerning the code base for such a system. Would it be better (or even feasible) to write a single set of code that would adequately work with the varying sensor systems and mobility features of each platform? Or would it be better to have two completely different sets of routines for each type of robot?

An interesting aspect of a heterogeneous population of robots is the varying capabilities of each type of robot. While the NOMAD 200 has a more precise positioning capability, the NOMAD SCOUT is somewhat smaller and may be able to explore spaces the NOMAD 200 cannot reach. It would be interesting to find a way to use the diversity of the robot types as an advantage in accurate and complete exploration and map making. Both NRL and NPS now have both types of robots, thus providing a basis for such work. Some theoretical work has already been done in this area [Ref. 36].

7. Identifying System Tradeoffs

Initial research has already identified some of the tradeoffs of optimizing or adjusting various aspects of the system. For instance, reducing the "trustworthy" sonar range results in much less problems with specular reflections, but increases the total distance a robot or number of robots must travel in order to map a given area. This

increased travel distance, especially the larger number of small movements necessary to sweep the sonar sensors at new frontiers, leads to increased odometry error. On the other hand, increasing the "trusted" sonar range reduces travel requirements, but causes a corresponding increase in false sonar returns.

Much more study is needed on optimizing the sensor model for the best combination of accurate mapping with minimal movement. Of course an adequate localization routine would solve many problems, but even with a good localization routine minimizing travel distances would be beneficial to the overall system. Other possible tradeoff studies include autonomy versus centralized control and heterogeneous versus homogeneous robot populations. Some work has already been done in the study of the interaction between quantity of robots in a system, sensor quality, and mobility constraints on system performance for a given mission [Ref. 37].

8. Modified Movement Behaviors

The original movement behaviors for the robot in the routine *robot.cc* were written with the capabilities of the NOMAD 200 robot in mind. Problems arise with using these same behaviors on the NOMAD SCOUT robot. While the NOMAD 200 can translate on its own axis, the NOMAD SCOUT cannot. Therefore movement behaviors that would have been simple rotations or shallow arcs on a NOMAD 200 become much larger movements when the conversion macros interpret them for use on the NOMAD SCOUT.

This causes many difficulties when the NOMAD SCOUT is near to and facing a wall or other obstacle at the end of a sensor sweep. When the robot tries to move on to a

new frontier the macro translates the current movement commands into a forward turning motion. If the robot is too close to the wall it will be blocked and eventually the new frontier it should have traveled to will be marked as inaccessible. Some type of backing up or modified turn behavior would seem to be a possible solution.

Other opportunities also exist to counter the "Follow The Leader" and "Dancing Robot" behaviors. One possible solution to be explored might be to have the robot broadcast its location either to all robots in the system or at least those nearby. That way a robot could recognize that the obstacle it is trying to map is in fact another robot. Of course any sort of centralized supervisory process could also counter this problem very easily. Another possibility is to set some sort of limit on how long a robot would continue to map a small area despite new frontiers constantly appearing in that area. After a period of time it could give up and move on to a radically different geographical area.

B.   HARDWARE AND SOFTWARE CHANGES

Other possible research areas would require additional hardware and/or modification of the already existing hardware in the system. In addition, software modifications would be necessary in order to use the added or changed hardware. Some of the hardware for the research possibilities mentioned below is already available at NPS.

1. Human – Robotic System Interaction

The area of real-time human-robotic interaction holds many, many possible research opportunities. In the current implementation the current map is displayed on a workstation and the opportunities for user interaction are very limited. Obviously, for a practical, deployable system these limitations must be removed.

There is a need for some sort of portable system through which a user can receive information from a single robot or groups of robots and also direct the actions of an individual robot or number of robots. Ideally, the device would be lightweight, unobtrusive, and user friendly in design and use. While earlier research had focused on rather large control and display systems [Ref. 38], perhaps the best model available today for such a device is the in the form factor and design of a Personal Digital Assistant (PDA).

There are a number of different types of PDAs available for research at NPS. Many of them have some type of wireless connection or network capability. What is required is a method for a map data to be transmitted to these devices in a useable format and some method for operator commands to be sent back to the network and then on to the robot(s). Once this is possible many other research opportunities become available. What is the best way to manage the system from the operator point-of-view? For a deployed system, how much control does the operator need or even want? Is controlling the robot(s) the operator's primary duty or something that should be done on an as-required basis? What are the possibilities for cooperative exploration of an area between

human and robot? How best should the system signal the operator to possible danger areas or other areas of interest? These and other questions should be investigated early before large amounts of funding are spent on programs that later are found to be unworkable or impractical to implement.

2. Outdoor Trials

Any practical, field deployable system will need to work outdoors as well indoors. Even if the system is build primarily for mapping the interiors of buildings the individual robots will most likely have to traverse rough, broken, urban terrain to travel from one operating site to another. Neither the NOMAD 200 nor the NOMAD SCOUT has much capability in this regard as currently configured. At this time their manufacturer has no announced plans to market any outdoor-capable models either.

However, there does exist other outdoor-capable robotic systems at NPS. In particular the "Shepherd" vehicle [Ref. 39, 40], under cooperative development by several different departments at NPS, is a robot with a four-wheeled all-terrain-vehicle (ATV) style chassis with independent driving and steering capability. Much study has been conducted on this platform concerning motion control and localization via inertial sensors and the use of a Global Positioning System (GPS) receiver. It would seem that integrating some sort of mapping capability into it would be a natural next step. There are many questions about how much of the current implementation of the frontier-based exploration code could be ported to the new platform, but the fundamentals of the process would seem to remain the same.

3. Removing Dependency on Wired Network

Perhaps one of the most ambitious possibilities is removing the dependency that the current implementation has on a wired network to provide communications connectivity. The NOMAD SCOUT has the capability to be completely independent through the use of a laptop computer running the LINUX operating system. A laptop can be mounted on top of the NOMAD SCOUT and can run the same code locally (after it is recompiled) that is currently run on a remote Sun workstation.

Once the NOMAD SCOUT robots are operating independently of a wired network it might be possible to better implement a broadcast model of communication amongst the robots. The wireless modems used in the current implementation are capable of serving as either point-to-point communications stations or as part of a distributed system. Because there would no longer be a shared memory location, the exploration and mapping software would have to be modified to actually send all the map data and not just a pointer to the data or message that it is available to the other robots in the system. There is already a body of work supporting communications protocols for distributed robotic systems without a centralized communications server [Ref. 41].

4. Additional Sensor Systems

Using laptops on the NOMAD SCOUT robots as mentioned above also opens up the opportunity to integrate additional sensor systems on the platform. The unused input ports of the laptop provide a means to include video, audio, or any of a number of

other sensing devices to the system. In addition, there is also the possibility of adding a means of detecting beacons in the environment or on other robots as an aid to navigation and localization.

VIII. CONCLUSIONS

Oftentimes the hardest part of any journey is just getting started. This thesis and the research involved in creating it have been aimed at creating a starting point for future studies. Now that the basics of a real world (as compared to simulated) multiple robot system has been developed and implemented at NPS, a huge number of additional avenues of investigation are available.

It has been shown that much work remains in order to create a consistent and robust exploration and mapping system before many of the questions surrounding robotic battlefield support can be answered. However, now there exists at NPS a "critical mass" of mobile robot types with varying capabilities and at least basic software to enable some of them to operate in a shared real-world environment toward a common goal.

Many of the problems encountered in this research are very similar to those discovered by other researchers when moving a robotic system from one environment to another [Ref. 42]. In the case of this research the testing of multiple mobile robots has been moved from simulation-only environment to a combination of simulation and real-world testing. This transition has revealed many details of multiple robotic systems that otherwise would have remained hidden in simulator-only testing.

In Chapter VI there are listed many possible areas of study based on the work presented here. These are just the start of many possible thesis opportunities involving hardware, software, human-machine interaction, etc. One of the most exciting and challenging things about robotics as a field of study is the number of different fields and

disciplines that it encompasses. It is hoped that future students will take up this

challenge and carry on the work started here.

This appendix contains the source code that was used to collect sonar range data on early map making efforts with the NOMAD SCOUT robot.

```
1    /*****************************************************************
2     *
3     * PROGRAM: world_sonar.c
4     *
5     * PURPOSE: To collect sonar data for establishing a world map.
6     * modified for Scout by Patrick A. Hillmeyer
7     *****************************************************************/
8
9
10   /*** Include Files ***/
11
12   #include "Nclient.h"
13   #include <stdio.h>
14   #include.<stdlib.h>
15   #include <math.h>
16
17
18
19   /***  Conversion MACROS courtesy of Nomadic Inc   ***/
20   /** original beta macros for SCOUT models  ****/
21
22   #define RIGHT(trans, steer)   (trans + (int)((float)steer*368.61/3600.0))
23   #define LEFT(trans, steer)    (trans - (int)((float)steer*368.61/3600.0))
24
25   #define scout_vm(trans, steer)   vm(RIGHT(trans, steer), LEFT(trans,
26   steer), 0)
27   #define scout_pr(trans, steer)   pr(RIGHT(trans, steer), LEFT(trans,
28   steer), 0)
29
30
31
32
33   /*** Function Prototypes ***/
34
35   void GetSensorData(void);
36
37
38   /*** Globals ***/
39
40   long SonarRange[ 16] ;  /* array of sonar readings (inches) */
41   long IRRange[ 16] ;  /* array of infrared readings (no units) */
42   long robot_config[ 4] ;
43
44   /*** Main Program ***/
45
46   main (unsigned int argc, char** argv)
47   {
48      int i, j, index;
49      int order[ 16] ;
50      FILE *fp;
51
```

```
52      /* Connect to Nserver. The parameter passed must always be 1. */
53      connect_robot(1, MODEL_SCOUT, "scout1.ece.nps.navy.mil", 4001);
54
55
56      /* Initialize Smask and send to robot. Smask is a large array that
57      controls which data the robot returns back to the server. This
58      function tells the robot to give us everything. */
59      init_mask();
60
61
62      /* Configure timeout (given in seconds). This is how long the robot
63      will keep moving if you become disconnected. Set this low if there
64      are walls nearby. */
65      conf_tm(1);
66
67
68      /* Sonar setup  */
69      for (i = 0; i < 16; i++)
70        order[ i]  = i;
71      conf_sn(15,order);
72
73
74      zr();   /* tell robot to zero itself */
75
76
77
78      fp = fopen("range.dat", "w");
79
80      /* Main loop. */
81      for (i=0; i<2; i++)
82        {
83          GetSensorData();
84
85          for (j=0; j<16; j++)
86          fprintf(fp, "%8d %8d %8d %8d %8d  \n",
87                  robot_config[ 0] ,robot_config[ 1] ,robot_config[ 2] ,
88                  robot_config[ 3] ,
89                  SonarRange[ j] );
90        }
91
92      fclose(fp);
93
94      /* Disconnect. */
95      disconnect_robot(1);
96   }
97
98
99
100   /* GetSensorData(). Read in sensor data and load into arrays. */
101   void GetSensorData (void)
102   {
103     int i;
104
105
106     /* Read all sensors and load data into State array. */
107     gs();
108
109
```

130

```
110    /* Read State array data and put readings into individual arrays. */
111    for (i = 0; i < 16; i++)
112      {
113        /* Sonar ranges are given in inches, and can be between 6 and
114        255, inclusive. */
115        SonarRange[ i] = State[ 17+i] ;
116
117        /* IR readings are between 0 and 15, inclusive. This value is
118        inversely proportional to the light reflected by the detected
119        object, and is thus proportional to the distance of the
120        object. Due to the many environmental variables effecting the
121        reflectance of infrared light, distances cannot be accurately
122        ascribed to the IR readings. */
123        IRRange[ i] = State[ 1+i] ;
124      }
125
126
127    for (i = 0; i < 4; i++)
128      robot_config[ i] = State[ 34+i] ;
129  }
```

APPENDIX B. MATLAB SOURCE CODE FOR PLOTTING OF SIMPLE SENSOR RETURN DATA

This appendix contains the MATLAB M-file that was used to analyze and display the sonar range data from early map making efforts with the NOMAD SCOUT robot.

```
1    % Capt Patrick A. Hillmeyer, USMC
2    % Code originally written for EC 4300 Robotics class
3    % Written to interpret sonar range data collected
4    % from a NOMAD SCOUT robot
5
6    % Load the range data collected during the robot's travel
7    load range1.dat
8
9    robo_data=range1;
10
11   % Convert robot x,y coordinates to inches
12   rob_x_in_world=robo_data(:,1)/10;
13   rob_y_in_world=robo_data(:,2)/10;
14
15   % In this data the base and turret are aligned
16   % therefore no alignment correction is necessary
17   % Carryover from old NOMAD 200 version of code
18
19   % covert angles to degrees then to radians
20   base_angle=(robo_data(:,3)/10)*pi/180;
21   obj_dist_fr_rob=robo_data(:,5);
22
23   num_sensors=16;
24   deg_per_sensor=360/num_sensors;
25   rad_per_sensor=deg_per_sensor*pi/180;
26
27   % correct for sensor location offset
28   % from robot center
29   rob_radius=8.81;
30
31   % Set the range at which to trust the
32   % sonar data
33   s_trust=60;
34
35   % plot robot path alone
36   figure(1)
37   plot(rob_x_in_world,rob_y_in_world,'w.')
38   title('Robot path in real (or simulated) world')
39   xlabel('Inches'),ylabel('Inches')
40   axis('equal')
41
42   % now plot sonar hits as robot moved
43   x_sonar_hits=[];
44   y_sonar_hits=[];
45
46   % Read through the data in sets corresponding
47   % to the number of sensor readings taken at each
48   % location in the robot's path
49   for ctr2=0:((length(robo_data)/num_sensors)-1)
50
51     for ctr3=1:num_sensors
```

```
52
53     abs_data_pt=(num_sensors*ctr2)+ctr3;
54
55     % only process if valid reading
56     if obj_dist_fr_rob(abs_data_pt)<s_trust
57
58       A_B_T=[ cos(base_angle(abs_data_pt)) ...
59              -sin(base_angle(abs_data_pt)) ...
60              0 rob_x_in_world(abs_data_pt);
61              sin(base_angle(abs_data_pt)) ...
62              cos(base_angle(abs_data_pt)) ...
63              0 rob_y_in_world(abs_data_pt);
64              0 0 1 0;
65              0 0 0 1];
66
67       % correct for off-by-one discrepency
68       % in sensor numbering
69       sensor_num=ctr3-1;
70
71       B_P=[ (obj_dist_fr_rob(abs_data_pt)+rob_radius) ...
72              *cos(sensor_num*rad_per_sensor);
73              (obj_dist_fr_rob(abs_data_pt)+rob_radius) ...
74              *sin(sensor_num*rad_per_sensor);
75              0;
76              1];
77
78       A_P=A_B_T*B_P;
79
80       x_sonar_hits=[ x_sonar_hits A_P(1)];
81       y_sonar_hits=[ y_sonar_hits A_P(2)];
82
83     end  % end for if
84
85    end % end for ctr3
86
87   end % end for ctr2
88
89   figure(2)
90   plot(x_sonar_hits,y_sonar_hits,'w.', ...
91        rob_x_in_world,rob_y_in_world,'w.')
92   title('Simulated world sonar data - 60 inch sonar reliability')
93   xlabel('Inches'),ylabel('Inches')
94   axis('equal')
```

This appendix contains the header file for the routine that builds the evidence grid based on the sensor return data.

```
1    /*
2
3      grid.h
4
5      Header file for robot/evidence grid functions
6      original code by Brian Yamauchi
7
8      Modifications for SCOUT THESIS
9      by Patrick A. Hillmeyer
10
11   */
12
13   #include "cmacs.h"
14   #include "volsense.h"
15
16   /* Grid occupied threshold */
17
18   #define GRID_POS_THRESH 16
19
20   /* Grid unoccupied threshold */
21
22   #define GRID_NEG_THRESH -16
23
24   /* Local Grid dimensions (feet) */
25
26   /* BEGIN SCOUT THESIS CHANGE */
27   /* change the local grid dimensions to match the global grid dimensions
28   */
29   #define X_MIN -22.0
30   #define X_MAX 22.0
31   #define Y_MIN -22.0
32   #define Y_MAX 22.0
33   #define Z_MIN 0.0
34   #define Z_MAX 5.0
35
36   /* Grid resolution (cells) */
37   /* increase the number of cells  */
38   /* the value here has to be a power of 2 and symetrical  */
39   /*   i.e 64 by 64, 128 by 128, etc    */
40   /* this is true for all the other grid resolutions below as well */
41
42   #define X_RES 256
43   #define Y_RES 256
44   #define Z_RES 1
45
46   /* END SCOUT THESIS CHANGE  */
47
48
49   /* Global grid dimensions (feet) */
50
51   #define GLOBAL_X_MIN -22.0
52   #define GLOBAL_X_MAX 22.0
```

```
53   #define GLOBAL_Y_MIN -22.0
54   #define GLOBAL_Y_MAX 22.0
55   #define GLOBAL_Z_MIN 0.0
56   #define GLOBAL_Z_MAX 5.0
57
58
59
60   /* Global grid resolution (cells) */
61
62   #define GLOBAL_X_RES 256
63   #define GLOBAL_Y_RES 256
64   #define GLOBAL_Z_RES 1
65
66
67
68   /* Navigation grid dimensions (feet) */
69
70   #define NAV_X_MIN -22.0
71   #define NAV_X_MAX 22.0
72   #define NAV_Y_MIN -22.0
73   #define NAV_Y_MAX 22.0
74   #define NAV_Z_MIN 0.0
75   #define NAV_Z_MAX 5.0
76
77
78
79   /* Resolution of navigation grid (cells) */
80
81   #define NAV_X_RES 256
82   #define NAV_Y_RES 256
83   #define NAV_Z_RES 1
84
85
86
87   /* Sensor modes */
88
89   #define SONAR_MODE 0
90   #define LASER_MODE 1
91   #define INTEG_MODE 2
92
93   /* Sensor parameters */
94
95   /* BEGIN SCOUT THESIS CHANGE  */
96
97   /* Scout and Scout2 dimensions 15.125 in sensor to sensor diameter  */
98   /* Scout2 sonar height 10.25 in  Scout close enough to use same value */
99   /* Height from floor to sonar (ft)  Scouts 10.25 in */
100  #define SONAR_HEIGHT 0.8542
101  /* Offset from robot center to sonar (ft)  Scouts 7.5625 in */
102  #define SONAR_RAD 0.63
103  /* Separation between adjacent sonars (deg)  - same as Nomad 200 */
104  #define SONAR_SEP 22.5
105  /* Height from floor to IR (ft)  - None on Scout */
106  #define IR_HEIGHT 0.0
107  /* Offset from robot center to IR (ft) - None on Scout */
108  #define IR_RAD 0.0
109  /* Separation between adjacent IR (deg) - None on Scout */
110  #define IR_SEP 0.0
```

```c
111    /* Height from floor to laser (ft) - None on Scout */
112    #define LASER_HEIGHT 0.0
113    /*  END SCOUT THESIS CHANGE  */
114
115     #define HEIGHT_OFFSET 0.0    /* z-axis offset (ft) */
116
117    /* Maximum sonar reading (indicates no reflection) */
118    #define MAX_SONAR_READING 255
119
120    /* Maximum (valid) sonar range (feet) -- Use 21.25 for no truncation */
121
122    /* BEGIN SCOUT THESIS CHANGE  */
123
124    /* This is the trustworthy range of the sonar in ft */
125    /* shorter range for Scout to reduce specular reflection problem */
126    #define MAX_SONAR_RANGE 8.0
127
128    /*#define MAX_SONAR_RANGE 10.0*/
129    /*#define MAX_SONAR_RANGE 21.25*/
130
131    /* Maximum sonar range for occupied cells (feet) */
132
133    /* This value seems to have no effect */
134    /*#define MAX_SONAR_OCC_RANGE 3.0*/
135    #define MAX_SONAR_OCC_RANGE 15.0
136
137    /* Maximum IR reading (indicates no reflection) */
138
139    #define MAX_IR_READING 0     /* No IR on Scout */
140
141    /* Maximum (valid) laser range (feet) */
142
143    /*#define MAX_LASER_RANGE 100.0*/
144    #define MAX_LASER_RANGE 0.0    /* No laser on Scout */
145
146    /* END SCOUT THESIS CHANGE */
147
148    /* Size of cell in robot window */
149
150    #define DISPLAY_SCALE 56.25
151
152    /* Angle conversion constants */
153
154    #define M_RAD2DEG 57.29578
155    #define M_DEG2RAD 0.017453293
156
157    /* Laser configuration parameters */
158
159    #define LASER_MODE_OFF 0x32    /* 1 100 1 1 */   /* X,Y pairs */
160    #define LASER_MODE_ON 0x33     /* 1 100 1 1 */   /* X,Y pairs */
161    #define LINE 0x03              /* 0 000 1 1 */   /* X,Y pairs for endpoints
162    */
163    #define THRESHHOLD 70     /* f4=30, f2.8=5,  factory=20 */
164    #define WIDTH 40          /* f4=20, f2.8=20, factory=20 */
165    #define NUMDATA 120       /* Number of points returned */
166    #define AVG 1             /* Number of pixels averaged */
167
168    /* Stepsize for printing grid */
```

```
169
170     #define PRINT_STEP 1
171
172
173     /* Robot size */
174
175     /* BEGIN SCOUT THESIS CHANGE */
176     /* Scout2 is taller use its value - 14 in */
177     /* Add bumper space for total radius */
178
179     /* Robot radius (feet)  Scout sensor radius plus .756 in for bumpers*/
180     #define ROBOT_RADIUS 0.693
181
182     /* Robot height (feet)  use Scout2 14 in */
183     #define ROBOT_HEIGHT 1.1667
184
185     /* Size necessary for safe robot passage (feet) */
186     #define ROBOT_PASSAGE_RADIUS 0.7     /* Add small safety margin */
187
188     /* END SCOUT THESIS CHANGE */
189
190
191     /* Grid decay factor */
192
193     #define GRID_DECAY 8
194
195     /* Grid translation parameters */
196
197     #define NUM_TRANS 1               /* Number of translations in each
198     direction
199                                         along each axis */
200     #define TRANS_STEP 0.2           /* Size of each translation step (feet) */
201
202     /* Grid rotation parameters */
203
204     #define NUM_ROT 1        /* Number of rotations (in each direction) */
205     #define ROT_STEP 2.0            /* Rotation step (degrees) */
206
207     /* Mimimum change in position (1/10 inch) to update */
208
209     #define MIN_DELTA 46.88
210
211     /* Relative weight of clear cells in fine grid to coarse grid conversion
212     */
213
214     #define F2C_CLEAR_WT 1
215
216     /* Relative weight of occupied cells in fine grid to coarse grid
217     conversion */
218
219     #define F2C_OCC_WT 4
220
221     /* Maximum laser/sonar angle difference for laser-limited sonar
222     (degrees) */
223
224     #define LLS_MAX_ANGLE_DIFF 3.0
```

APPENDIX D. FRONTIER-BASED EXPLORATION CODE – GRID.C

This appendix contains the source code for the routine that builds the evidence grid based on the sensor return data.

```
1    /*
2
3      grid.c
4
5      Robot/evidence grid functions
6      original code by Brian Yamauchi
7
8      Modification for SCOUT THESIS
9      by Patrick A. Hillmeyer
10
11   */
12
13   #include <stdio.h>
14   #include <math.h>
15   #include "Nclient.h"
16   #include "grid.h"
17
18   double min3(double x, double y, double z)
19   /*
20     Return the minimum of three values
21    */
22   {
23       double m;       /* Minimum */
24
25       m = x;
26       if (y < m) {
27          m = y;
28       }
29       if (z < m) {
30          m = z;
31       }
32       return(m);
33   }
34
35   int world2grid(Map3D map, double wx, double wy, double wz,
36                  int *gx, int *gy, int *gz)
37   /*
38     Return grid coordinates for location in world coordinates
39    */
40   {
41       double xsize, ysize, zsize;           /* Size of grid cell */
42
43       if ((wx < map.lomv[ 0] ) || (wx > map.himv[ 0] ) ||
44          (wy < map.lomv[ 1] ) || (wy > map.himv[ 1] ) ||
45          (wz < map.lomv[ 2] ) || (wz > map.himv[ 2] )) {
46   /*    printf("world2grid: point (%f, %f, %f) out of range <%f:%f, %f:%f,
47   %f:%f>.\n",
48                  wx, wy, wz, map.lomv[ 0] , map.himv[ 0] , map.lomv[ 1] ,
49   map.himv[ 1] ,
50                  map.lomv[ 2] , map.himv[ 2] );*/
51          return(-1);
52       }
```

```
53
54      xsize = (map.himv[ 0] - map.lomv[ 0] ) / map.msize[ 0] ;
55      ysize = (map.himv[ 1] - map.lomv[ 1] ) / map.msize[ 1] ;
56      zsize = (map.himv[ 2] - map.lomv[ 2] ) / map.msize[ 2] ;
57
58      *gx = (int) ((wx - map.lomv[ 0] ) / xsize);
59      *gy = (int) ((wy - map.lomv[ 1] ) / ysize);
60      *gz = (int) ((wz - map.lomv[ 2] + HEIGHT_OFFSET) / zsize);
61
62      if ((*gx < 0) || (*gx >= map.msize[ 0] ) ||
63         (*gy < 0) || (*gy >= map.msize[ 1] ) ||
64         (*gz < 0) || (*gz >= map.msize[ 2] )) {
65  /*     printf("world2grid: world location (%f, %f, %f) --> cell [ %d, %d,
66  %d] out of range.\n", wx, wy, wz, *gx, *gy, *gz);*/
67         return(-1);
68      }
69
70      return(1);
71  }
72
73  int world2index(Map3D map, double wx, double wy, double wz)
74  /*
75    Return grid cell index for location in world coordinates
76   */
77  {
78      double xsize, ysize, zsize;          /* Size of grid cell */
79      int gx, gy, gz;                      /* Coordinates of grid cell */
80      int index;                           /* Grid cell array index */
81
82      if ((wx < map.lomv[ 0] ) || (wx > map.himv[ 0] ) ||
83         (wy < map.lomv[ 1] ) || (wy > map.himv[ 1] ) ||
84         (wz < map.lomv[ 2] ) || (wz > map.himv[ 2] )) {
85  /*     printf("world2index (%f, %f, %f) out of range <%f:%f, %f:%f,
86  %f:%f>.\n",          wx, wy, wz, map.lomv[ 0] , map.himv[ 0] , map.lomv[ 1] ,
87  map.himv[ 1] ,
88                 map.lomv[ 2] , map.himv[ 2] );*/
89         return(-1);
90      }
91
92      xsize = (map.himv[ 0] - map.lomv[ 0] ) / map.msize[ 0] ;
93      ysize = (map.himv[ 1] - map.lomv[ 1] ) / map.msize[ 1] ;
94      zsize = (map.himv[ 2] - map.lomv[ 2] ) / map.msize[ 2] ;
95
96      gx = (int) ((wx - map.lomv[ 0] ) / xsize);
97      gy = (int) ((wy - map.lomv[ 1] ) / ysize);
98      gz = (int) ((wz - map.lomv[ 2] + HEIGHT_OFFSET) / zsize);
99
100     index = gz * map.msize[ 0] * map.msize[ 1] + gy * map.msize[ 0] + gx;
101
102     if ((index < 0) || (index >= map.msize[ 0] * map.msize[ 1] *
103  map.msize[ 2] )) {
104 /*     printf("world2index: world location (%f, %f, %f) --> index [ %d]
105  out of range.\n", wx, wy, wz, index);*/
106         return(-1);
107     }
108
109 /*     printf("world2grid: world location (%f, %f, %f) --> cell [ %d, %d,
110  %d] <%d>.\n", wx, wy, wz, gx, gy, gz, index);*/
```

```
111         fflush(stdout);
112
113         return(index);
114     }
115
116     void grid2world(Map3D map, int gx, int gy, int gz,
117                     double *wx, double *wy, double *wz)
118     /*
119       Return world coordinates for location in grid coordinates
120     */
121     {
122         double xsize, ysize, zsize;            /* Size of grid cell */
123     /*     int tx, ty, tz;*/
124
125         xsize = (map.himv[ 0] - map.lomv[ 0] ) / map.msize[ 0] ;
126         ysize = (map.himv[ 1] - map.lomv[ 1] ) / map.msize[ 1] ;
127         zsize = (map.himv[ 2] - map.lomv[ 2] ) / map.msize[ 2] ;
128
129         *wx = (double) (gx + 0.5) * xsize + map.lomv[ 0] ;
130         *wy = (double) (gy + 0.5) * ysize + map.lomv[ 1] ;
131         *wz = (double) (gz + 0.5) * zsize + map.lomv[ 2] ;
132
133     /*     if (world2grid(map, *wx, *wy, *wz, &tx, &ty, &tz) == -1) {
134             printf("<%d, %d, %d> --> (%f, %f, %f) --> <???, ???, ????>\n",
135                     gx, gy, gz, *wx, *wy, *wz);
136         }
137         else {
138           printf("<%d, %d, %d> --> (%f, %f, %f) --> <%d, %d, %d>\n",
139                     gx, gy, gz, *wx, *wy, *wz, tx, ty, tz);
140         }*/
141     }
142
143     int grid2index(Map3D map, int gx, int gy, int gz)
144     /*
145       Return grid cell index for grid cell coordinates
146     */
147     {
148         int index;                          /* Grid cell array index */
149
150         index = gz * map.msize[ 0] * map.msize[ 1] + gy * map.msize[ 0] + gx;
151         return(index);
152     }
153
154     void set_location(Map3D map, double x, double y, double z, int value)
155     /*
156        Set probability of grid cell corresponding to world location
157     */
158     {
159         int gindex;                 /* Grid array index */
160
161         gindex = world2index(map, x, y, z);
162         if (gindex > -1) {
163           map.mapm[ gindex] = value;
164         }
165     }
166
167     void set_grid(Map3D map, int x, int y, int z, int value)
168     /*
```

```
169          Set probability of specified grid cell
170      */
171      {
172          int gindex;                    /* Grid array index */
173
174          gindex = z * map.msize[ 0] * map.msize[ 1] + y * map.msize[ 0] + x;
175          if ((gindex < 0) || (gindex >= map.msize[ 0] * map.msize[ 1] *
176      map.msize[ 2] )) {
177      /*     printf("set_grid: cell [ %d, %d, %d] out of range <%d, %d, %d>.\n",
178                  x, y, z, map.msize[ 0] , map.msize[ 1] , map.msize[ 2] );*/
179          return;
180          }
181          map.mapm[ gindex] = value;
182      }
183
184      void grid_init(Map3D *map1,     /* Grid pointer */
185                  double cx, /* Center x-coord (feet) */
186                  double cy) /* Center y-coord (feet) */
187      /*
188        Initialize evidence grid
189      */
190      {
191        double lov[ 3] , hiv[ 3] ;       /* Grid corners (feet) */
192        int msize[ 3] ;                  /* Grid size (cells) */
193
194        map1->cx = cx;
195        map1->cy = cy;
196
197        msize[ 0] = X_RES;
198        lov[ 0] = cx + X_MIN;
199        hiv[ 0] = cx + X_MAX;
200
201        msize[ 1] = Y_RES;
202        lov[ 1] = cy + Y_MIN;
203        hiv[ 1] = cy + Y_MAX;
204
205        msize[ 2] = Z_RES;
206        lov[ 2] = Z_MIN;
207        hiv[ 2] = Z_MAX;
208
209        MakeMap3D(msize, lov, hiv, map1);
210      }
211
212      void grid_init_global(Map3D *map1,   /* Grid pointer */
213                      double cx,   /* Center x-coord (feet) */
214                      double cy)   /* Center y-coord (feet) */
215      /*
216        Initialize global evidence grid
217      */
218      {
219        double lov[ 3] , hiv[ 3] ;       /* Grid corners (feet) */
220        int msize[ 3] ;                  /* Grid size (cells) */
221
222        map1->cx = cx;
223        map1->cy = cy;
224
225        msize[ 0] = GLOBAL_X_RES;
226        lov[ 0] = cx + GLOBAL_X_MIN;
```

```
227     hiv[ 0]  = cx + GLOBAL_X_MAX;
228
229     msize[ 1]  = GLOBAL_Y_RES;
230     lov[ 1]  = cy + GLOBAL_Y_MIN;
231     hiv[ 1]  = cy + GLOBAL_Y_MAX;
232
233     msize[ 2]  = GLOBAL_Z_RES;
234     lov[ 2]  = GLOBAL_Z_MIN;
235     hiv[ 2]  = GLOBAL_Z_MAX;
236
237     MakeMap3D(msize, lov, hiv, map1);
238   }
239
240   void grid_init_nav(Map3D *map1,      /* Grid pointer */
241                      double cx,       /* Center x-coord (feet) */
242                      double cy)       /* Center y-coord (feet) */
243   /*
244      Initialize evidence grid for navigation
245   */
246   {
247     double lov[ 3] , hiv[ 3] ;       /* Grid corners (feet) */
248     int msize[ 3] ;                  /* Grid size (cells) */
249
250     map1->cx = cx;
251     map1->cy = cy;
252
253     msize[ 0]  = NAV_X_RES;
254     lov[ 0]  = cx + NAV_X_MIN;
255     hiv[ 0]  = cx + NAV_X_MAX;
256
257     msize[ 1]  = NAV_Y_RES;
258     lov[ 1]  = cy + NAV_Y_MIN;
259     hiv[ 1]  = cy + NAV_Y_MAX;
260
261     msize[ 2]  = NAV_Z_RES;
262     lov[ 2]  = NAV_Z_MIN;
263     hiv[ 2]  = NAV_Z_MAX;
264
265     MakeMap3D(msize, lov, hiv, map1);
266   }
267
268   void grid_print(Map3D map, int yaxis)
269   /*
270      Print evidence grid occupancy probabilities
271   */
272   {
273     int x, y, z;                     /* Cell index */
274     int xsize, ysize, zsize;      /* Grid dimensions (# cells) */
275     int p;                       /* Occupancy probability */
276     int empty;                       /* Empty level flag */
277
278     xsize = map.msize[ 0] ;
279     ysize = map.msize[ 1] ;
280     zsize = map.msize[ 2] ;
281
282     for (z = 0; z < zsize; z++) {
283       y = x = 0;
284       empty = 1;
```

```
285        while((y < ysize) && (x < xsize) && empty) {
286          if (map.mapm[ z * xsize * ysize + y * xsize + x] != 0) {
287          empty = 0;
288          }
289          x++;
290          if (x == xsize) {
291          x = 0;
292          y++;
293          }
294        }
295
296        if (!empty) {
297          printf("Level: %d\n\n", z);
298
299          for (y = 0; y < ysize; y++) {
300          for (x = 0; x < xsize; x++) {
301            if (yaxis == 1) {
302              p = map.mapm[ z * xsize * ysize + (ysize - y - 1) * xsize + x];
303            }
304            else {
305              p = map.mapm[ z * xsize * ysize + y * xsize + x];
306            } .
307            if (p > 0) {
308              printf("#");
309            }
310            else if (p == 0) {
311              printf("?");
312            }
313            else if (p > -25) {
314              printf(":");
315            }
316            else if (p > -50) {
317              printf(".");
318            }
319            else {
320              printf(" ");
321            }
322          }
323          printf("\n");
324          }
325          getchar();
326        }
327      }
328    }
329
330    void sonar_print(Map3D map, int yaxis)
331    /*
332      Print evidence grid occupancy probabilities for sonar level
333    */
334    {
335      int x, y, z;                    /* Cell index */
336      int xsize, ysize, zsize;    /* Grid dimensions (# cells) */
337      int p;                      /* Occupancy probability */
338      int empty;                      /* Empty level flag */
339
340      xsize = map.msize[ 0] ;
341      ysize = map.msize[ 1] ;
342      zsize = map.msize[ 2] ;
```

```
343
344     z = (int) ((SONAR_HEIGHT + HEIGHT_OFFSET - map.lomv[ 2] ) /
345               (map.himv[ 2] - map.lomv[ 2] ) * zsize);
346
347     printf("");
348     for (y = 0; y < ysize; y += PRINT_STEP) {
349       for (x = 0; x < xsize; x += PRINT_STEP) {
350         if (yaxis == 1) {
351         p = map.mapm[ z * xsize * ysize + (ysize - y - 1) * xsize + x] ;
352         }
353         else {
354         p = map.mapm[ z * xsize * ysize + y * xsize + x] ;
355         }
356         if (p > 0) {
357         printf("#");
358         }
359         else if (p == 0) {
360         printf("?");
361         }
362         else if (p > -25) {
363         printf(":");
364         }
365         else if (p > -50) {
366         printf(".");
367         }
368         else {
369         printf(" ");
370         }
371       }
372       printf("\n");
373     }
374   }
375
376   void laser_print(Map3D map, int yaxis)
377   /*
378      Print evidence grid occupancy probabilities for sonar level
379   */
380   {
381     int x, y, z;                    /* Cell index */
382     int xsize, ysize, zsize;     /* Grid dimensions (# cells) */
383     int p;                   /* Occupancy probability */
384     int empty;                      /* Empty level flag */
385
386     xsize = map.msize[ 0] ;
387     ysize = map.msize[ 1] ;
388     zsize = map.msize[ 2] ;
389
390     z = (int) ((LASER_HEIGHT + HEIGHT_OFFSET - map.lomv[ 2] ) /
391               (map.himv[ 2] - map.lomv[ 2] ) * zsize);
392
393     printf("");
394     for (y = 0; y < ysize; y += PRINT_STEP) {
395       for (x = 0; x < xsize; x += PRINT_STEP) {
396         if (yaxis == 1) {
397         p = map.mapm[ z * xsize * ysize + (ysize - y - 1) * xsize + x] ;
398         }
399         else {
400         p = map.mapm[ z * xsize * ysize + y * xsize + x] ;
```

```
401        }
402        if (p > 0) {
403        printf("#");
404        }
405        else if (p == 0) {
406        printf("?");
407        }
408        else if (p > -25) {
409        printf(":");
410        }
411        else if (p > -50) {
412        printf(".");
413        }
414        else {
415        printf(" ");
416        }
417      }
418    printf("\n");
419    }
420 }
421
422 void grid_display(Map3D map,          /* Evidence grid */
423                 double height,  /* z-coord of plane to display */
424                 int x_origin,   /* World x-coord of origin (1/10 inch)*/
425                 int y_origin)   /* World y-coord of origin (1/10 inch)*/
426 /*
427    Display evidence grid occupancy probabilities in robot window
428 */
429 {
430    double xd, yd;                    /* Display coords */
431    double xscale, yscale, zscale;    /* Cell dimensions (tenths of
432 inches) */
433    double xoffset, yoffset;          /* Circle center offset */
434    int x, y, z;                      /* Cell index */
435    int xsize, ysize, zsize;          /* Grid dimensions (# cells) */
436    int p;                          /* Occupancy probability */
437    int empty;                        /* Empty level flag */
438 /*  int rad;*/                        /* Radius of cell display */
439
440    printf("Displaying grid at (%d, %d)\n", x_origin, y_origin);
441
442    xsize = map.msize[0];
443    ysize = map.msize[1];
444    zsize = map.msize[2];
445
446    xscale = (map.himv[0] - map.lomv[0]) * 120.0 / (double) xsize;
447    yscale = (map.himv[1] - map.lomv[1]) * 120.0 / (double) ysize;
448    zscale = (map.himv[2] - map.lomv[2]) * 120.0 / (double) zsize;
449
450    xoffset = xscale / 2.0;
451    yoffset = yscale / 2.0;
452
453    z = (int) ((height + HEIGHT_OFFSET - map.lomv[2]) /
454            (map.himv[2] - map.lomv[2]) * zsize);
455
456    for (y = 0; y < ysize; y++) {
457      for (x = 0; x < xsize; x++) {
458        p = map.mapm[z * xsize * ysize + y * xsize + x];
```

```
459
460          xd = (int) ((double) x * xscale + map.lomv[ 0] * 120.0) + x_origin;
461          yd = (int) ((double) y * yscale + map.lomv[ 1] * 120.0) + y_origin;
462
463   /*       rad = (int) (((double) (p - NEG) / (double) POS) * (double)
464   xscale * 0.5);
465
466          draw_arc(xd, yd, rad, rad, 0, 3600, 1);*/
467
468          if (p > 0) {
469             draw_arc(xd, yd, xscale, yscale, 0, 3600, 1);
470          }
471          else if (p == 0) {
472             draw_arc(xd, yd, xscale / 4.0, xscale / 4.0, 0, 3600, 1);
473          }
474       }
475     }
476   }
477
478   void grid_display_pos(Map3D map,              /* Evidence grid */
479                      double height)    /* z-coord of plane to display */
480   /*
481      Display evidence grid occupancy probabilities in robot window at
482   position
483   */
484   {
485      int dx, dy;
486
487      printf("Enter display coordinates ==> ");
488      scanf(" %d %d", &dx, &dy);
489
490      grid_display(map, height, dx, dy);
491   }
492
493   void model_init(CylSensorModelArray *sonar_smd,
494                   CylSensorModelArray *sonar_clear_smd)
495   /*
496      Initialize sensor models
497   */
498   {
499      InitCylModelParams();
500
501      MakeCylModel(66, 0.02, 64, 128, 1.0, 22.0, sonar_smd);
502      TrimCylModel(*sonar_smd);
503
504   /*   WriteCylModel(*sonar_smd, "sonar.mod");*/
505   /*   ReadCylModel("sonar.mod", sonar_smd);*/
506
507      MakeClearCylModel(66, 0.02, 64, 128, 1.0, 22.0, sonar_clear_smd);
508      TrimCylModel(*sonar_clear_smd);
509
510   /*   WriteCylModel(*sonar_clear_smd, "clear.mod");*/
511   /*   ReadCylModel("clear.mod", sonar_clear_smd);*/
512   }
513
514   void sonar_scan(CylSensorModelArray smd, CylSensorModelArray clear_smd,
515                   Map3D map, int rx, int ry, int rtheta)
516   /*
```

```
517        Update evidence grid using all sonar sensors
518    */
519    {
520        PosData sonar_pose[ 16] ;        /* Sonar pose information */
521        double robot_x, robot_y;        /* Robot position */
522        double robot_theta;             /* Robot heading */
523        double range;                   /* Range reading (feet) */
524        double angle;                   /* Sensor angle (radians) */
525        double sonar_pos[ 3] ;          /* Sonar position */
526        double sonar_dir[ 3] ;          /* Sonar direction */
527        int reading;                    /* Raw sonar reading */
528        int i;                   /* Sonar index */
529
530        gs();   /* SCOUT THESIS CHANGE use gs to get sonar and position info */
531
532    /*  posSonarRingGet(sonar_pose);   SCOUT THESIS CHANGE - comment this
533    line out  */
534    /*  SCOUT does not currently provide pose data as NOMAD 200 does  */
535
536        for (i = 0; i < 16; i++) {
537    /* SCOUT THESIS CHANGE
538    comment out the requests for pose information below
539    robot_x = (double) sonar_pose[ i] .config.configX / 120.0;  commented out
540    robot_y = (double) sonar_pose[ i] .config.configY / 120.0;  commented out
541    robot_theta = (double) sonar_pose[ i] .config.configTurret / 10.0;
542    commented out
543    */
544
545    /* SCOUT THESIS CHANGE uncomment out the lines below
546    and get the sonar data from using the gs command */
547        robot_x = (double) rx / 120.0;
548        robot_y = (double) ry / 120.0;
549        robot_theta = (double) rtheta / 10.0;
550
551        reading = State[ i + 17] ;
552        range = (double) reading / 12.0;
553        angle = ((double) i * SONAR_SEP + robot_theta) * M_DEG2RAD;
554
555        sonar_dir[ 0]  = cos(angle);
556        sonar_dir[ 1]  = sin(angle);
557        sonar_dir[ 2]  = 0.0;
558
559        sonar_pos[ 0]  = sonar_dir[ 0]  * SONAR_RAD;
560        sonar_pos[ 1]  = sonar_dir[ 1]  * SONAR_RAD;
561        sonar_pos[ 2]  = SONAR_HEIGHT + HEIGHT_OFFSET;
562
563        if ((reading != MAX_SONAR_READING) && (range <= MAX_SONAR_RANGE)){
564           AddCylReading(range, sonar_pos, sonar_dir, smd, map);
565        }
566        else {
567           AddCylReading(MAX_SONAR_RANGE, sonar_pos, sonar_dir, clear_smd,
568    map);
569        }
570     }
571    }
572
573    void sonar_scan_abs(CylSensorModelArray smd, CylSensorModelArray
574    clear_smd,
```

```
575                     Map3D map, int rx, int ry, int rtheta)
576     /*
577        Update evidence grid using all sonar sensors (using absolute position)
578     */
579     {
580        PosData sonar_pose[ 16] ;        /* Sonar pose information */
581        double robot_x, robot_y;        /* Robot position */
582        double robot_theta;             /* Robot heading */
583        double range;                   /* Range reading (feet) */
584        double angle;                   /* Sensor angle (radians) */
585        double sonar_pos[ 3] ;          /* Sonar position */
586        double sonar_dir[ 3] ;          /* Sonar direction */
587        int reading;                    /* Raw sonar reading */
588        int i;                    /* Sonar index */
589
590        gs();
591     /*  posSonarRingGet(sonar_pose);  SCOUT THESIS CHANGE - comment this
592     line out */
593     /* SCOUT does not currently provide for pose data as the NOMAD 200 does
594     */
595
596        for (i = 0; i < 16; i++) {
597     /*     robot_x = (double) sonar_pose[ i] .config.configX / 120.0;  **
598     comment this line out */
599     /*     robot_y = (double) sonar_pose[ i] .config.configY / 120.0;   **
600     comment out   */
601     /*     robot_theta = (double) sonar_pose[ i] .config.configTurret / 10.0;
602     **   comment out */
603
604     /* uncomment out the lines below and use gs command to get sonar data
605     */
606        robot_x = (double) rx / 120.0;
607        robot_y = (double) ry / 120.0;
608        robot_theta = (double) rtheta / 10.0;
609
610        reading = State[ i + 17] ;
611        range = (double) reading / 12.0;
612        angle = ((double) i * SONAR_SEP + robot_theta) * M_DEG2RAD;
613
614        sonar_dir[ 0]  = cos(angle);
615        sonar_dir[ 1]  = sin(angle);
616        sonar_dir[ 2]  = 0.0;
617
618        sonar_pos[ 0]  = sonar_dir[ 0]  * SONAR_RAD + robot_x;
619        sonar_pos[ 1]  = sonar_dir[ 1]  * SONAR_RAD + robot_y;
620        sonar_pos[ 2]  = SONAR_HEIGHT + HEIGHT_OFFSET;
621
622        if ((reading != MAX_SONAR_READING) && (range <= MAX_SONAR_RANGE)){
623          AddCylReading(range, sonar_pos, sonar_dir, smd, map);
624        }
625        else {
626          AddCylReading(MAX_SONAR_RANGE, sonar_pos, sonar_dir, clear_smd,
627     map);
628        }
629     }
630     }
631
632
```

```
633
634     /* BEGIN SCOUT CHANGE */
635     /* NOTE - it appears that the following function is never used by any
636     exploration routine  */
637     /* Left in code for now  since it will not affect the Scout  */
638     /* The header for this is in grid++.h  */
639
640     void ir_scan_abs(CylSensorModelArray smd, CylSensorModelArray clear_smd,
641                    Map3D map, int rx, int ry, int rtheta)
642     /*
643        Update evidence grid using all infrared sensors (using absolute
644     position)
645     */
646     {
647        PosData ir_pose[ 16] ;          /* IR pose information */
648        double robot_x, robot_y;      /* Robot position */
649        double robot_theta;           /* Robot heading */
650        double range;                 /* Range reading (feet) */
651        double angle;                 /* Sensor angle (radians) */
652        double ir_pos[ 3] ;           /* IR position */
653        double ir_dir[ 3] ;           /* IR direction */
654        int reading;                  /* Raw IR reading */
655        int i;                    /* Sonar index */
656
657        gs();
658        posInfraredRingGet(ir_pose);
659
660        for (i = 0; i < 16; i++) {
661           robot_x = (double) ir_pose[ i] .config.configX / 120.0;
662           robot_y = (double) ir_pose[ i] .config.configY / 120.0;
663           robot_theta = (double) ir_pose[ i] .config.configTurret / 10.0;
664
665           /*      robot_x = (double) rx / 120.0;
666           robot_y = (double) ry / 120.0;
667           robot_theta = (double) rtheta / 120.0;*/
668
669           reading = State[ i + 17] ;
670           range = (double) reading / 12.0;
671           angle = ((double) i * IR_SEP + robot_theta) * M_DEG2RAD;
672
673           ir_dir[ 0] = cos(angle);
674           ir_dir[ 1] = sin(angle);
675           ir_dir[ 2] = 0.0;
676
677           ir_pos[ 0] = ir_dir[ 0] * IR_RAD + robot_x;
678           ir_pos[ 1] = ir_dir[ 1] * IR_RAD + robot_y;
679           ir_pos[ 2] = IR_HEIGHT + HEIGHT_OFFSET;
680
681           if (reading < MAX_IR_READING) {
682              AddCylReading(range, ir_pos, ir_dir, smd, map);
683           }
684        }
685     }
686
687     /* END SCOUT CHANGE */
688
689
690     void sonar_scan_abs_norep(CylSensorModelArray smd,
```

150

```
691                          CylSensorModelArray clear_smd,
692                          Map3D map, int rx, int ry, int rtheta)
693     /*
694        Update evidence grid using all sonar sensors (using absolute position)
695        (no updates for repeated positions)
696     */
697     {
698        static long old_x[ 16] , old_y[ 16] ; /* Old robot position */
699        static int first_flag = 1;  /* Reset first time function is called */
700
701        PosData sonar_pose[ 16] ;               /* Sonar pose information */
702        double robot_x, robot_y;      /* Robot position */
703        double delta;                 /* Change in robot position since last
704     update */
705        double robot_theta;           /* Robot heading */
706        double range;                 /* Range reading (feet) */
707        double angle;                 /* Sensor angle (radians) */
708        double sonar_pos[ 3] ;        /* Sonar position */
709        double sonar_dir[ 3] ;        /* Sonar direction */
710        int reading;                  /* Raw sonar reading */
711        int i;                   /* Sonar index */
712
713        gs ();
714        posSonarRingGet (sonar_pose);
715
716        for (i = 0; i < 16; i++) {
717           robot_x = (double) .sonar_pose[ i] .config.configX / 120.0;
718           robot_y = (double) sonar_pose[ i] .config.configY / 120.0;
719           robot_theta = (double) sonar_pose[ i] .config.configTurret / 10.0;
720
721           delta = hypot ((double) (sonar_pose[ i] .config.configX - old_x[ i] ),
722                     (double) (sonar_pose[ i] .config.configY - old_y[ i] ));
723
724           if (first_flag || delta >= MIN_DELTA) {
725              old_x[ i]  = sonar_pose[ i] .config.configX;
726              old_y[ i]  = sonar_pose[ i] .config.configY;
727
728              reading = State[ i + 17] ;
729              range = (double) reading / 12.0;
730              angle = ((double) i * SONAR_SEP + robot_theta) * M_DEG2RAD;
731
732              sonar_dir[ 0]  = cos (angle);
733              sonar_dir[ 1]  = sin (angle);
734              sonar_dir[ 2]  = 0.0;
735
736              sonar_pos[ 0]  = sonar_dir[ 0]  * SONAR_RAD + robot_x;
737              sonar_pos[ 1]  = sonar_dir[ 1]  * SONAR_RAD + robot_y;
738              sonar_pos[ 2]  = SONAR_HEIGHT + HEIGHT_OFFSET;
739
740              if ((reading != MAX_SONAR_READING) && (range <= MAX_SONAR_RANGE)){
741              AddCylReading(range, sonar_pos, sonar_dir, smd, map);
742              }
743              else {
744              AddCylReading(MAX_SONAR_RANGE, sonar_pos, sonar_dir, clear_smd,
745     map);
746              }
747           }
748           /*     else {
```

151

```
749            printf("sonar_scan_abs_norep: Repeated position (%d, %d) for
750  sensor %d.\n",
751            old_x[i], old_y[i], i);
752      }*/
753    }
754
755    first_flag = 0;
756  }
757
758  void laser_update(Map3D map, double rx, double ry, double lx, double ly,
759                  double rtheta)
760  /*
761    Update evidence grid for a single laser reading
762   */
763  {
764      double lr, ltheta;              /* Laser vector */
765      double wx, wy;                  /* World coords of laser endpoint */
766      double xsize, ysize;            /* Size of grid cell */
767      double stepsize;                /* Stepsize along laser axis */
768      double dx, dy;                  /* Stepsize along x and y axes */
769      double px, py;                  /* Point currently being updated */
770      int steps;                      /* Number of steps */
771      int i;
772
773      lr = hypot(lx, ly);
774      ltheta = atan2(ly, lx) * M_RAD2DEG;
775
776      wx = rx + lr * cos((ltheta + rtheta) * M_DEG2RAD);
777      wy = ry + lr * sin((ltheta + rtheta) * M_DEG2RAD);
778
779      set_location(map, wx, wy, LASER_HEIGHT, POS);
780
781      px = rx;
782      py = ry;
783
784      xsize = (map.himv[0] - map.lomv[0]) / map.msize[0];
785      ysize = (map.himv[1] - map.lomv[1]) / map.msize[1];
786
787      if (xsize < ysize) {
788        stepsize = xsize;
789      }
790      else {
791        stepsize = ysize;
792      }
793
794      dx = stepsize * cos((ltheta + rtheta) * M_DEG2RAD);
795      dy = stepsize * sin((ltheta + rtheta) * M_DEG2RAD);
796
797      steps = (int) (lr / stepsize);
798
799      for (i = 0; i < steps; i++) {
800        set_location(map, px, py, LASER_HEIGHT, NEG);
801        px += dx;
802        py += dy;
803      }
804  }
805
806  void laser_scan(Map3D map, int rx, int ry, int rtheta)
```

```
807     /*
808        Update evidence grid using laser scanner
809     */
810     {
811        PosData laser_pose;          /* Laser pose information */
812        double lx, ly, lr, ltheta;   /* Laser point (robot coordinates) */
813        double wx, wy, wz;           /* Laser point (world coordinates) */
814        double robot_x, robot_y, robot_theta;   /* Robot location */
815        int i;
816
817        gs();
818        posLaserGet(&laser_pose);
819
820        robot_x = (double) laser_pose.config.configX / 120.0;
821        robot_y = (double) laser_pose.config.configY / 120.0;
822        robot_theta = (double) laser_pose.config.configTurret / 10.0;
823
824        for (i = 0; i < Laser[0]; i++) {
825           /*        printf("[ %d, %d] ", Laser[i * 2 + 1], Laser[i * 2 + 2]);*/
826           if (Laser[i * 2 + 1] != 65000) {
827              lx = (double) Laser[i * 2 + 1] / 120.0;
828              ly = (double) Laser[i * 2 + 2] / 120.0;
829              /*     printf("(%f, %f)", lx, ly);*/
830
831     /*        laser_update(map, robot_x, robot_y, lx, ly, robot_theta);*/
832
833              lr = hypot(lx, ly);
834              ltheta = atan2(ly, lx) * M_RAD2DEG;
835
836              if (lr <= MAX_LASER_RANGE) {
837                 wx = lr * cos((ltheta + robot_theta) * M_DEG2RAD);
838                 wy = lr * sin((ltheta + robot_theta) * M_DEG2RAD);
839                 wz = LASER_HEIGHT + HEIGHT_OFFSET;
840                 set_location(map, wx, wy, wz, POS);
841                 /*            draw_line((int) robot_x * 120.0, (int) robot_y *
842     120.0, (int) wx * 120.0,
843                          (int) wy * 120.0, 19);*/
844              }
845           }
846     /*        printf("\n");*/
847        }
848     }
849
850     void laser_scan_abs(Map3D map, int rx, int ry, int rtheta)
851     /*
852        Update evidence grid using laser scanner (using absolute position)
853     */
854     {
855        PosData laser_pose;          /* Laser pose information */
856        double lx, ly, lr, ltheta;   /* Laser point (robot coordinates) */
857        double wx, wy, wz;           /* Laser point (world coordinates) */
858        double robot_x, robot_y, robot_theta;   /* Robot location */
859        int i;
860
861        gs();
862        posLaserGet(&laser_pose);
863
864        robot_x = (double) laser_pose.config.configX / 120.0;
```

```
865     robot_y = (double) laser_pose.config.configY / 120.0;
866     robot_theta = (double) laser_pose.config.configTurret / 10.0;
867
868     for (i = 0; i < Laser[ 0] ; i++) {
869       /*          printf("[ %d, %d] ", Laser[ i * 2 + 1] , Laser[ i * 2 + 2] );*/
870         if (Laser[ i * 2 + 1]  != 65000) {
871           lx = (double) Laser[ i * 2 + 1]  / 120.0;
872           ly = (double) Laser[ i * 2 + 2]  / 120.0;
873           /*     printf(" (%f, %f)", lx, ly);*/
874
875   /*          laser_update(map, robot_x, robot_y, lx, ly, robot_theta);*/
876
877           lr = hypot(lx, ly);
878           ltheta = atan2(ly, lx) * M_RAD2DEG;
879
880           if (lr <= MAX_LASER_RANGE) {
881             wx = lr * cos((ltheta + robot_theta) * M_DEG2RAD) + robot_x;
882             wy = lr * sin((ltheta + robot_theta) * M_DEG2RAD) + robot_y;
883             wz = LASER_HEIGHT + HEIGHT_OFFSET;
884             set_location(map, wx, wy, wz, POS);
885             /*              draw_line((int) robot_x * 120.0, (int) robot_y *
886   120.0, (int) wx * 120.0,
887                            (int) wy * 120.0, 19);*/
888           }
889         }
890   /*       printf("\n");*/
891       }
892   }
893
894   double laser_min(void)
895   /*
896     Return minimum laser range reading
897   */
898   {
899     double min_range = MAX_LASER_RANGE;      /* Minimum range reading */
900     double lx, ly, lr, ltheta;              /* Laser point (robot coordinates)
901   */
902     int i;
903
904     gs();
905
906     for (i = 0; i < Laser[ 0] ; i++) {
907       if (Laser[ i * 2 + 1]  != 65000) {
908         lx = (double) Laser[ i * 2 + 1]  / 120.0;
909         ly = (double) Laser[ i * 2 + 2]  / 120.0;
910
911         lr = hypot(lx, ly);
912
913         if (lr < min_range) {
914         min_range = lr;
915         }
916       }
917     }
918
919     return(min_range);
920   }
921
922   void lls_scan(CylSensorModelArray smd, CylSensorModelArray clear_smd,
```

```
923                     Map3D map, int rx, int ry, int rtheta)
924     /*
925        Update evidence grid using laser-limited sonar
926     */
927
928     {
929        PosData sonar_pose[ 16] ;       /* Sonar pose information */
930        PosData ir_pose[ 16] ;          /* IR pose information */
931        PosData laser_pose;             /* Laser pose information */
932        double sonar_x, sonar_y;        /* Sonar position */
933        double sonar_theta;             /* Sonar angle */
934        double laser_x, laser_y;        /* Laser position */
935        double laser_theta;             /* Laser angle */
936        double lx, ly, lr, ltheta;      /* Laser point (robot coordinates) */
937        double wx, wy, wz;              /* Laser point (world coordinates) */
938        double min_laser_range = MAX_LASER_RANGE;       /* Minimum laser reading
939     */
940        double sonar_range;             /* Range reading (feet) */
941        double angle;                   /* Sensor angle (radians) */
942        double sonar_pos[ 3] ;          /* Sonar position */
943        double sonar_dir[ 3] ;          /* Sonar direction */
944        double·angle_diff;              /* Angle offset between laser and sonar */
945        int reading;                    /* Raw sonar reading */
946        int i;
947
948        /* Get sensor and pose data from robot */
949
950        gs();
951        posSonarRingGet(sonar_pose);
952        posInfraredRingGet(ir_pose);
953        posLaserGet(&laser_pose);
954
955        /* Update grid using laser readings */
956
957        laser_x = (double) laser_pose.config.configX / 120.0;
958        laser_y = (double) laser_pose.config.configY / 120.0;
959        laser_theta = (double) laser_pose.config.configTurret / 10.0;
960
961        for (i = 0; i < Laser[ 0] ; i++) {
962           /*        printf("[ %d, %d]  ", Laser[ i * 2 + 1], Laser[ i * 2 + 2] );*/
963              if (Laser[ i * 2 + 1]  != 65000) {
964                 lx = (double) Laser[ i * 2 + 1] / 120.0;
965                 ly = (double) Laser[ i * 2 + 2] / 120.0;
966                 /*     printf(" (%f, %f)", lx, ly);*/
967
968     /*        laser_update(map, laser_x, laser_y, lx, ly, laser_theta);*/
969
970                 lr = hypot(lx, ly);
971                 if (lr < min_laser_range) {
972                    min_laser_range = lr;
973                 }
974
975                 ltheta = atan2(ly, lx) * M_RAD2DEG;
976
977                 if (lr <= MAX_LASER_RANGE) {
978                    wx = lr * cos((ltheta + laser_theta) * M_DEG2RAD);
979                    wy = lr * sin((ltheta + laser_theta) * M_DEG2RAD);
980                    wz = LASER_HEIGHT + HEIGHT_OFFSET;
```

155

```
981              set_location(map, wx, wy, wz, POS);
982              /*         draw_line((int) laser_x * 120.0, (int) laser_y *
983    120.0, (int) wx * 120.0,
984                         (int) wy * 120.0, 19);*/
985          }
986      }
987   /*      printf("\n");*/
988    }
989
990    /* Update grid using sonar reading (limited by minimum laser range) */
991
992    sonar_x = (double) sonar_pose[ 0] .config.configX / 120.0;
993    sonar_y = (double) sonar_pose[ 0] .config.configY / 120.0;
994    sonar_theta = (double) sonar_pose[ 0] .config.configTurret / 10.0;
995
996    reading = State[ 17] ;
997
998    /* At very close ranges, use infrared instead */
999
1000   /*   if (State[ 1] < MAX_IR_READING) {
1001        reading = State[ 1] ;
1002
1003        sonar_x = (double) ir_pose[ 0] .config.configX / 120.0;
1004        sonar_y = (double) ir_pose[ 0] .config.configY / 120.0;
1005        sonar_theta = (double) ir_pose[ 0] .config.configTurret / 10.0;
1006
1007        printf("laser/IR offset = %f inches : %f degrees\n",
1008        hypot(sonar_x - laser_x, sonar_y - laser_y),
1009        sonar_theta - laser_theta);
1010        }
1011        else {
1012        printf("laser/sonar offset = %f inches : %f degrees\n",
1013        hypot(sonar_x - laser_x, sonar_y - laser_y),
1014        sonar_theta - laser_theta);
1015   }*/
1016
1017   /* Compute angle offset between laser and sonar (or IR) */
1018
1019   angle_diff = fabs(sonar_theta - laser_theta);
1020   if (angle_diff > 180.0) {
1021      angle_diff = 360.0 - angle_diff;
1022   }
1023
1024   /* Discard reading if offset is too large */
1025
1026   if (angle_diff > LLS_MAX_ANGLE_DIFF) {
1027      printf("LLS reading discarded: angle offset = %f\n", angle_diff);
1028      return;
1029   }
1030
1031   /* Determine LLS range */
1032
1033   sonar_range = (double) reading / 12.0;
1034
1035   if (sonar_range > min_laser_range) {
1036      sonar_range = min_laser_range;
1037   }
1038
```

```
1039        /* Update grid */
1040
1041        angle = sonar_theta * M_DEG2RAD;
1042
1043        sonar_dir[ 0] = cos(angle);
1044        sonar_dir[ 1] = sin(angle);
1045        sonar_dir[ 2] = 0.0;
1046
1047        sonar_pos[ 0] = sonar_dir[ 0] * SONAR_RAD;
1048        sonar_pos[ 1] = sonar_dir[ 1] * SONAR_RAD;
1049        sonar_pos[ 2] = SONAR_HEIGHT + HEIGHT_OFFSET;
1050
1051        if ((reading != MAX_SONAR_READING) && (sonar_range <=
1052     MAX_SONAR_RANGE)){
1053            if (sonar_range <= MAX_SONAR_OCC_RANGE) {
1054                AddCylReading(sonar_range, sonar_pos, sonar_dir, smd, map);
1055            }
1056            else {
1057                AddCylReading(sonar_range, sonar_pos, sonar_dir, clear_smd, map);
1058            }
1059        }
1060        else {
1061            AddCylReading(MAX_SONAR_RANGE, sonar_pos, sonar_dir, clear_smd,
1062     map);
1063        }
1064     }
1065
1066     void lls_scan_abs(CylSensorModelArray smd, CylSensorModelArray
1067     clear_smd,
1068                     Map3D map, int rx, int ry, int rtheta)
1069     /*
1070        Update evidence grid using laser-limited sonar (absolute coordinates)
1071     */
1072
1073     {
1074        PosData sonar_pose[ 16] ;        /* Sonar pose information */
1075        PosData ir_pose[ 16] ;          /* IR pose information */
1076        PosData laser_pose;             /* Laser pose information */
1077        double sonar_x, sonar_y;        /* Sonar position */
1078        double sonar_theta;             /* Sonar angle */
1079        double laser_x, laser_y;        /* Laser position */
1080        double laser_theta;             /* Laser angle */
1081        double lx, ly, lr, ltheta;      /* Laser point (robot coordinates) */
1082        double wx, wy, wz;              /* Laser point (world coordinates) */
1083        double min_laser_range = MAX_LASER_RANGE;      /* Minimum laser reading
1084     */
1085        double sonar_range;             /* Range reading (feet) */
1086        double angle;                   /* Sensor angle (radians) */
1087        double sonar_pos[ 3] ;          /* Sonar position */
1088        double sonar_dir[ 3] ;          /* Sonar direction */
1089        double angle_diff;              /* Angle offset between laser and sonar */
1090        int reading;                    /* Raw sonar reading */
1091        int i;
1092
1093        /* Get sensor and pose data from robot */
1094
1095        gs();
1096        posSonarRingGet(sonar_pose);
```

```
1097        posInfraredRingGet(ir_pose);
1098        posLaserGet(&laser_pose);
1099
1100        /* Update grid using laser readings */
1101
1102        laser_x = (double) laser_pose.config.configX / 120.0;
1103        laser_y = (double) laser_pose.config.configY / 120.0;
1104        laser_theta = (double) laser_pose.config.configTurret / 10.0;
1105
1106        for (i = 0; i < Laser[ 0] ; i++) {
1107          /*        printf("[ %d, %d] ", Laser[ i * 2 + 1] , Laser[ i * 2 + 2] );*/
1108            if (Laser[ i * 2 + 1] != 65000) {
1109              lx = (double) Laser[ i * 2 + 1] / 120.0;
1110              ly = (double) Laser[ i * 2 + 2] / 120.0;
1111              /*    printf(" (%f, %f)", lx, ly);*/
1112
1113   /*        laser_update(map, laser_x, laser_y, lx, ly, laser_theta);*/
1114
1115              lr = hypot(lx, ly);
1116              if (lr < min_laser_range) {
1117                min_laser_range = lr;
1118              }.
1119
1120              ltheta = atan2(ly, lx) * M_RAD2DEG;
1121
1122              if (lr <= MAX_LASER_RANGE) {
1123                wx = lr * cos((ltheta + laser_theta) * M_DEG2RAD) + laser_x;
1124                wy = lr * sin((ltheta + laser_theta) * M_DEG2RAD) + laser_y;
1125                wz = LASER_HEIGHT + HEIGHT_OFFSET;
1126                set_location(map, wx, wy, wz, POS);
1127                /*              draw_line((int) laser_x * 120.0, (int) laser_y *
1128   120.0, (int) wx * 120.0,
1129                        (int) wy * 120.0, 19);*/
1130              }
1131            }
1132   /*        printf("\n");*/
1133      }
1134
1135      /* Update grid using sonar reading (limited by minimum laser range) */
1136
1137      sonar_x = (double) sonar_pose[ 0] .config.configX / 120.0;
1138      sonar_y = (double) sonar_pose[ 0] .config.configY / 120.0;
1139      sonar_theta = (double) sonar_pose[ 0] .config.configTurret / 10.0;
1140
1141      reading = State[ 17] ;
1142
1143      /* At very close ranges, use infrared instead */
1144
1145      /*  if (State[ 1] < MAX_IR_READING) {
1146        reading = State[ 1] ;
1147
1148        sonar_x = (double) ir_pose[ 0] .config.configX / 120.0;
1149        sonar_y = (double) ir_pose[ 0] .config.configY / 120.0;
1150        sonar_theta = (double) ir_pose[ 0] .config.configTurret / 10.0;
1151
1152        printf("IR/sonar offset = %f inches : %f degrees\n",
1153        hypot(sonar_x - laser_x, sonar_y - laser_y),
1154        sonar_theta - laser_theta);
```

```
1155      }
1156      else {
1157      printf("laser/sonar offset = %f inches : %f degrees\n",
1158      hypot(sonar_x - laser_x, sonar_y - laser_y),
1159      sonar_theta - laser_theta);
1160      }*/
1161
1162      /* Compute angle offset between laser and sonar (or IR) */
1163
1164      angle_diff = fabs(sonar_theta - laser_theta);
1165      if (angle_diff > 180.0) {
1166        angle_diff = 360.0 - angle_diff;
1167      }
1168
1169      /* Discard reading if offset is too large */
1170
1171      if (angle_diff > LLS_MAX_ANGLE_DIFF) {
1172        printf("LLS reading discarded: angle offset = %f\n", angle_diff);
1173        return;
1174      }
1175
1176      /* Determine LLS range */
1177
1178      sonar_range = (double) reading / 12.0;
1179
1180      if (sonar_range > min_laser_range) {
1181        sonar_range = min_laser_range;
1182      }
1183
1184      /* Update grid */
1185
1186      angle = sonar_theta * M_DEG2RAD;
1187
1188      sonar_dir[ 0] = cos(angle);
1189      sonar_dir[ 1] = sin(angle);
1190      sonar_dir[ 2] = 0.0;
1191
1192      sonar_pos[ 0] = sonar_dir[ 0] * SONAR_RAD + sonar_x;
1193      sonar_pos[ 1] = sonar_dir[ 1] * SONAR_RAD + sonar_y;
1194      sonar_pos[ 2] = SONAR_HEIGHT + HEIGHT_OFFSET;
1195
1196      if ((reading != MAX_SONAR_READING) && (sonar_range <=
1197  MAX_SONAR_RANGE)){
1198        if (sonar_range <= MAX_SONAR_OCC_RANGE) {
1199          AddCylReading(sonar_range, sonar_pos, sonar_dir, smd, map);
1200        }
1201        else {
1202          AddCylReading(sonar_range, sonar_pos, sonar_dir, clear_smd, map);
1203        }
1204      }
1205      else {
1206        AddCylReading(MAX_SONAR_RANGE, sonar_pos, sonar_dir, clear_smd,
1207  map);
1208      }
1209  }
1210
1211  void clear_robot(Map3D map, int rx, int ry)
1212  {
```

```
1213        /* Set grid cells under robot to be unoccupied */
1214
1215        double wx, wy;                      /* Robot location (world/feet) */
1216        int lx, ly, lz, hx, hy, hz;            /* Corners of robot bounding
1217   box */
1218        int cx, cy, cz;                     /* Grid cell coordinates */
1219
1220
1221        wx = (double) rx / 120.0;
1222        wy = (double) ry / 120.0;
1223
1224        if (world2grid(map, wx - ROBOT_RADIUS, wy - ROBOT_RADIUS, 0.0,
1225                   &lx, &ly, &lz) == -1) {
1226          printf("clear_robot: robot edge (%f, %f, %f) out of range.\n",
1227                wx - ROBOT_RADIUS, wy - ROBOT_RADIUS, 0.0);
1228          return;
1229        }
1230        if (world2grid(map, wx + ROBOT_RADIUS, wy + ROBOT_RADIUS,
1231   ROBOT_HEIGHT,
1232                   &hx, &hy, &hz) == -1) {
1233          printf("clear_robot: robot edge (%f, %f, %f) out of range.\n",
1234                wx + ROBOT_RADIUS, wy + ROBOT_RADIUS, ROBOT_HEIGHT);
1235          return;
1236        }
1237
1238        for (cx = lx; cx <= hx; cx++) {
1239          for (cy = ly; cy <= hy; cy++) {
1240              for (cz = lz; cz <= hz; cz++) {
1241                 set_grid(map, cx, cy, cz, NEG);
1242              }
1243          }
1244        }
1245   }
1246
1247   void grid_clear(Map3D grid)
1248   /*
1249     Clear current grid;
1250    */
1251   {
1252        ClearMap3D(grid);
1253   }
1254
1255
1256   void grid_decay(Map3D grid)
1257   /*
1258     Decay grid cells towards base probability
1259    */
1260   {
1261        int k, km;
1262
1263        km =
1264   1<<(ILOG2C(grid.msize[0])+ILOG2C(grid.msize[1])+ILOG2C(grid.msize[2]));
1265
1266        for (k=0; k<km; k++) {
1267          if (grid.mapm[k] != 0) {
1268              if (grid.mapm[k] > GRID_DECAY) {
1269                 grid.mapm[k] -= GRID_DECAY;
1270              }
```

160

```
1271                else if (grid.mapm[ k] < -GRID_DECAY) {
1272                   grid.mapm[ k] += GRID_DECAY;
1273                }
1274                else {
1275                   grid.mapm[ k] = 0;
1276                }
1277           }
1278        }
1279    }
1280
1281    void grid_translate(Map3D grid1, Map3D grid2, double dx, double dy)
1282    /*
1283       Translate all cells in <grid1> by <dx, dy> (feet) and store results in
1284    <grid2>
1285     */
1286    {
1287        double wx, wy, wz;        /* World coords */
1288        int x, y, z;          /* Initial grid cell coordinates */
1289        int trans_index;         /* Transformed cell index */
1290
1291        printf("Translating by (%f, %f)\n", dx, dy);
1292
1293        printf("Initial grid\n");
1294        grid_display(grid1, SONAR_HEIGHT, 0.0, 0.0);
1295        printf("Hit <return>\n");
1296        getchar();
1297
1298        grid_clear(grid2);
1299
1300        for (x = 0; x < grid1.msize[ 0] ; x++) {
1301.         for (y = 0; y < grid1.msize[ 1] ; y++) {
1302             for (z = 0; z < grid1.msize[ 2] ; z++) {
1303                grid2world(grid1, x, y, z, &wx, &wy, &wz);
1304                trans_index = world2index(grid1, wx + dx, wy + dy, wz);
1305                if (trans_index != -1) {
1306                    grid2.mapm[ trans_index] =
1307                       grid1.mapm[ grid2index(grid1, x, y, z)] ;
1308                }
1309             }
1310          }
1311        }
1312
1313        printf("Translated grid\n");
1314        grid_display(grid2, SONAR_HEIGHT, 0.0, 0.0);
1315        printf("Hit <return>\n");
1316        getchar();
1317    }
1318
1319    void grid_rotate(Map3D grid1, Map3D grid2, double dtheta)
1320    /*
1321       Translate all cells in <grid1> by <dtheta> (degrees) and store results
1322    in
1323       <grid2>
1324     */
1325    {
1326        double wx, wy, wz;        /* Cartesian world coords of initial point
1327    */
1328        double rx, ry;           /* Rotated Cartesian coords */
```

161

```c
1329        double dx, dy;              /* Cartesian vector from center to point
1330 */
1331        double radtheta;            /* Rotation in radians */
1332        double r, theta;            /* Polar coords from center to point */
1333        int x, y, z;          /* Initial grid cell coordinates */
1334        int trans_index;            /* Transformed cell index */
1335
1336        printf("Rotating by (%f)\n", dtheta);
1337
1338        radtheta = dtheta * M_DEG2RAD;
1339
1340        printf("Initial grid\n");
1341        grid_display(grid1, SONAR_HEIGHT, 0.0, 0.0);
1342        printf("Hit <return>\n");
1343        getchar();
1344
1345        grid_clear(grid2);
1346
1347        for (x = 0; x < grid1.msize[ 0] ; x++) {
1348          for (y = 0; y < grid1.msize[ 1] ; y++) {
1349              for (z = 0; z < grid1.msize[ 2] ; z++) {
1350                  grid2world(grid1, x, y, z, &wx, &wy, &wz);
1351
1352                  dx = wx - grid1.cx;
1353                  dy = wy - grid1.cy;
1354
1355                  r = hypot(dy, dx);
1356                  theta = atan2(dy, dx);
1357
1358                  rx = grid1.cx + r * cos(theta + radtheta);
1359                  ry = grid1.cy + r * sin(theta + radtheta);
1360
1361                  trans_index = world2index(grid1, rx, ry, wz);
1362                  if (trans_index != -1) {
1363                      grid2.mapm[ trans_index] =
1364                          grid1.mapm[ grid2index(grid1, x, y, z)] ;
1365                  }
1366              }
1367          }
1368        }
1369
1370        printf("Rotated grid\n");
1371        grid_display(grid2, SONAR_HEIGHT, 0.0, 0.0);
1372        printf("Hit <return>\n");
1373        getchar();
1374 }
1375
1376
1377 double grid_match(Map3D stm, Map3D local)
1378 /*
1379   Match two (aligned) evidence grids
1380  */
1381 {
1382        double score = 0.0;         /* Match score */
1383        double wx, wy, wz;          /* World coords of match point */
1384        int x, y, z;          /* Grid cell coordinates of match point */
1385        int stm_index;              /* Index of cell in <stm> */
1386        int p1, p2;                 /* Corresponding probabilities */
```

```
1387        int m;                    /* Match point value (log scale) */
1388        int sum = 0;              /* Sum of match points values */
1389        int total = 0;            /* Total # of known points */
1390
1391        for (x = 0; x < local.msize[ 0] ; x++) {
1392           for (y = 0; y < local.msize[ 1] ; y++) {
1393               for (z = 0; z < local.msize[ 2] ; z++) {
1394                   p1 = local.mapm[ grid2index(local, x, y, z)] ;
1395                   grid2world(local, x, y, z, &wx, &wy, &wz);
1396                   stm_index = world2index(stm, wx, wy, wz);
1397                   if (stm_index != -1) {
1398                       p2 = stm.mapm[ stm_index] ;
1399                       total++;
1400                       if (((p1 < 0) && (p2 < 0)) ||
1401                           ((p1 > 0) && (p2 > 0)) ||
1402                           ((p1 == 0) && (p2 == 0))) {
1403                       sum++;
1404                       }
1405                   }
1406               }
1407           }
1408        }
1409
1410        score = (double) sum / (double) total;
1411  /*    score = (double) sum / (double) (local.msize[ 0] * local.msize[ 1] *
1412                               local.msize[ 2] );*/
1413  /*    printf("grid_match: sum = %d : score = %f\n", sum, score);*/
1414
1415        return(score);
1416  }
1417
1418  double trans_match(Map3D global, Map3D local, double *bx, double *by,
1419                 double *btheta)
1420  /*
1421    Transform and match two evidence grids
1422   */
1423  {
1424        Map3D local_t;            /* Translated local grid */
1425        Map3D local_tr;           /* Translated/rotated local grid */
1426        double score;         /* Current match score */
1427        double best_score;        /* Best match score over all transforms */
1428        double dx, dy;            /* Current translation distance */
1429        double dtheta;            /* Current rotation angle */
1430        double best_x, best_y;    /* Best current translation */
1431        double best_theta;        /* Best current rotation */
1432        double vx, vy, vtheta;    /* Transformation vector */
1433        int sx, sy;               /* Translation step counter */
1434        int stheta;               /* Rotation step counter */
1435
1436        grid_init(&local_t, local.cx, local.cy);
1437        grid_init(&local_tr, local.cx, local.cy);
1438
1439        vx = 0.0;
1440        vy = 0.0;
1441        vtheta = 0.0;
1442
1443        do {
1444           best_score = 0.0;
```

```
1445
1446            for (sx = -NUM_TRANS; sx <= NUM_TRANS; sx++) {
1447                for (sy = -NUM_TRANS; sy <= NUM_TRANS; sy++) {
1448                    for (stheta = -NUM_ROT; stheta <= NUM_ROT; stheta++) {
1449                        dx = TRANS_STEP * (double) sx;
1450                        dy = TRANS_STEP * (double) sy;
1451                        dtheta = ROT_STEP * (double) stheta;
1452                        grid_translate(local, local_t, vx + dx, vy + dy);
1453                        grid_rotate(local_t, local_tr, vtheta + dtheta);
1454                        score = grid_match(global, local_tr);
1455                        printf("translation (%f, %f) / rotation (%f): score =
1456   %f\n",
1457                            dx, dy, dtheta, score);
1458                        if (score > best_score) {
1459                          best_score = score;
1460                          best_x = dx;
1461                          best_y = dy;
1462                          best_theta = dtheta;
1463                        }
1464                    }
1465                }
1466            }
1467
1468        vx += best_x;
1469        vy += best_y;
1470        vtheta += best_theta;
1471
1472        printf("BEST translation (%f, %f) / rotation (%f): score = %f\n",
1473                best_x, best_y, best_theta, best_score);
1474
1475    }
1476    while((best_x != 0.0) || (best_y != 0.0) || (best_theta != 0.0));
1477
1478    *bx = vx;
1479    *by = vy;
1480    *btheta = vtheta;
1481
1482    return(best_score);
1483 }
1484
1485 void grid_copy(Map3D grid1, Map3D grid2)
1486 /*
1487    Copy <grid2> to <grid1>
1488  */
1489 {
1490     double wx, wy, wz;        /* World coords */
1491     int x, y, z;              /* Grid cell coordinates */
1492     int p;                    /* Cell occupancy probability */
1493
1494     for (x = 0; x < grid1.msize[0]; x++) {
1495       for (y = 0; y < grid1.msize[1]; y++) {
1496           for (z = 0; z < grid1.msize[2]; z++) {
1497             grid2world(grid1, x, y, z, &wx, &wy, &wz);
1498             grid1.mapm[grid2index(grid1, x, y, z)] =
1499                  grid2.mapm[world2index(grid2, wx, wy, wz)];
1500           }
1501       }
1502     }
```

```
1503    }
1504
1505    void grid_fine_to_coarse(Map3D fine, Map3D coarse)
1506    {
1507       double wx, wy, wz;           /* World coords */
1508       int x, y, z;                 /* Grid cell coordinates */
1509       int p;                  /* Cell occupancy probability */
1510       int findex;                  /* Index of cell in fine grid */
1511       int cindex;                  /* Index of cell in coarse grid */
1512       int cx, cy, cz;
1513
1514       grid_clear(coarse);
1515
1516       for (x = 0; x < fine.msize[ 0] ; x++) {
1517         for (y = 0; y < fine.msize[ 1] ; y++) {
1518            for (z = 0; z < fine.msize[ 2] ; z++) {
1519            findex = grid2index(fine, x, y, z);
1520
1521            grid2world(fine, x, y, z, &wx, &wy, &wz);
1522            cindex = world2index(coarse, wx, wy, wz);
1523
1524            world2grid(coarse, wx, wy, wz, &cx, &cy, &cz);
1525
1526            if (fine.mapm[ findex] < 0) {
1527               coarse.mapm[ cindex] -= F2C_CLEAR_WT;
1528            }
1529
1530            if (fine.mapm[ findex] > 0) {
1531               coarse.mapm[ cindex] += F2C_OCC_WT;
1532            }
1533
1534    //     if ((coarse.mapm[ cindex] == 0) ||
1535    //        (coarse.mapm[ cindex] < fine.mapm[ findex] )) {
1536    //      coarse.mapm[ cindex] = fine.mapm[ findex] ;
1537    //     }
1538
1539            }
1540         }
1541      }
1542    }
1543
1544    void integrate_grid(Map3D global,    /* Global grid */
1545                       Map3D local,  /* Local grid */
1546                       double lox,         /* Local x-origin (feet) */
1547                       double loy,         /* Local y-origin (feet) */
1548                       double lotheta)     /* Local origin rotation (degrees)
1549    */
1550    /*
1551       Integrate <local> grid within <global> grid
1552     */
1553    {
1554       /* Note: Assumes global origin is at (0,0,0) and only handles
1555          rotations in the xy-plane */
1556
1557       double lx, ly, lz;           /* Local coords (Cartesian) */
1558       double lr, ltheta;           /* Local coords (polar) */
1559       double wx, wy, wz;           /* World coords */
1560       double ix, iy, itheta;       /* Intermediate coords */
```

```
1561      int x, y, z;                   /* Grid cell coordinates */
1562      int p;                    /* Cell occupancy probability */
1563      int global_index;              /* Index of global grid cell */
1564      int local_index;               /* Index of local grid cell */
1565
1566      printf("integrate_grid: x = %f : y = %f : theta = %f\n", lox, loy,
1567    lotheta);
1568
1569      lotheta *= M_DEG2RAD;    /* Convert to radians */
1570
1571      for (x = 0; x < global.msize[ 0] ; x++) {
1572        for (y = 0; y < global.msize[ 1] ; y++) {
1573          for (z = 0; z < global.msize[ 2] ; z++) {
1574
1575            /* Convert cell index to global coords */
1576
1577            grid2world(global, x, y, z, &wx, &wy, &wz);
1578
1579            /* Convert global coords to local coords */
1580
1581            ix = wx - lox;
1582            iy = wy - loy;
1583            itheta = atan2(iy, ix);
1584
1585            lr = hypot(ix, iy);
1586            ltheta = itheta - lotheta;
1587
1588            lx = lr * cos(ltheta);
1589            ly = lr * sin(ltheta);
1590            lz = wz;           /* Assume z-coord is unchanged */
1591
1592            /*printf("global (%f, %f) --> local (%f, %f)\n", wx, wy, lx,
1593    ly);*/
1594
1595            /* Update global cell */
1596
1597            if ((lx >= X_MIN) && (lx <= X_MAX) &&
1598                (ly >= Y_MIN) && (ly <= Y_MAX) &&
1599                (lz >= Z_MIN) && (lz <= Z_MAX)) {
1600              global_index = grid2index(global, x, y, z);
1601              local_index = world2index(local, lx, ly, lz);
1602              global.mapm[ global_index] += local.mapm[ local_index] ;
1603
1604              if (global.mapm[ global_index] > POS) {
1605                global.mapm[ global_index]  = POS;
1606              }
1607              else if (global.mapm[ global_index] < NEG) {
1608                global.mapm[ global_index]  = NEG;
1609              }
1610            }
1611          }
1612        }
1613      }
1614    }
1615
1616    void integrate_global_grid(Map3D global,   /* Initial global grid */
1617                    Map3D global_new,    /* New global grid */
1618                    double nox,          /* Local x-origin (feet) */
```

166

```
1619                        double noy,          /* Local y-origin (feet) */
1620                        double notheta)      /* Local origin rotation (degrees)
1621   */
1622   /*
1623      Integrate <local> grid within <global> grid
1624    */
1625   {
1626      /* Note: Assumes global origin is at (0,0,0) and only handles
1627         rotations in the xy-plane */
1628
1629      double nx, ny, nz;           /* New global coords (Cartesian) */
1630      double nr, ntheta;           /* New Global coords (polar) */
1631      double wx, wy, wz;           /* World coords */
1632      double ix, iy, itheta;       /* Intermediate coords */
1633      int x, y, z;                 /* Grid cell coordinates */
1634      int p;                   /* Cell occupancy probability */
1635      int global_index;            /* Index of global grid cell */
1636      int new_index;           /* Index of new global grid cell */
1637
1638      printf("integrate_grid: x = %f : y = %f : theta = %f\n", nox, noy,
1639   notheta);
1640
1641      notheta *= M_DEG2RAD;    /* Convert to radians */
1642
1643      for (x = 0; x < global.msize[ 0] ; x++) {
1644         for (y = 0; y < global.msize[ 1] ; y++) {
1645            for (z = 0; z < global.msize[ 2] ; z++) {
1646
1647            /* Convert cell index to global coords */
1648
1649            grid2world(global, x, y, z, &wx, &wy, &wz);
1650
1651            /* Convert global coords to new global coords */
1652
1653            ix = wx - nox;
1654            iy = wy - noy;
1655            itheta = atan2(iy, ix);
1656
1657            nr = hypot(ix, iy);
1658            ntheta = itheta - notheta;
1659
1660            nx = nr * cos(ntheta);
1661            ny = nr * sin(ntheta);
1662            nz = wz;             /* Assume z-coord is unchanged */
1663
1664            /*printf("global (%f, %f) --> new global (%f, %f)\n", wx, wy, nx,
1665   ny);*/
1666
1667            /* Update global cell */
1668
1669            if ((nx >= GLOBAL_X_MIN) && (nx <= GLOBAL_X_MAX) &&
1670                (ny >= GLOBAL_Y_MIN) && (ny <= GLOBAL_Y_MAX) &&
1671                (nz >= GLOBAL_Z_MIN) && (nz <= GLOBAL_Z_MAX)) {
1672              global_index = grid2index(global, x, y, z);
1673              new_index = world2index(global_new, nx, ny, nz);
1674              global.mapm[ global_index]  += global_new.mapm[ new_index] ;
1675
1676              if (global.mapm[ global_index]  > POS) {
```

```
1677              global.mapm[ global_index]  = POS;
1678          }
1679          else if (global.mapm[ global_index] < NEG) {
1680              global.mapm[ global_index]  = NEG;
1681          }
1682        }
1683        }
1684      }
1685    }
1686  }
1687
1688  void save_grid(Map3D grid)              /* Evidence grid */
1689  /*
1690    Save evidence grid to file
1691   */
1692  {
1693      char filename[ 80] ;                /* Save file name */
1694
1695      printf("Enter save file name ==> ");
1696      scanf(" %s", filename);
1697
1698      printf("Saving grid to <%s>.\n", filename);
1699      WriteMap3D(grid, "Evidence Grid", "", filename);
1700  }
1701
1702  void save_grid_file(Map3D grid,            /* Evidence grid */
1703                  char *filename,      /* Save file name */
1704                  char *comment)       /* File header comment */
1705  /*
1706    Save evidence grid to specified file with header comment
1707   */
1708  {
1709    printf("Saving grid to <%s>.\n", filename);
1710    WriteMap3D(grid, comment, "", filename);
1711  }
1712
1713  void load_grid(Map3D *grid)             /* Evidence grid */
1714  /*
1715    Load evidence grid from file
1716   */
1717  {
1718      char filename[ 80] ;                /* Load file name */
1719      char title[ 80] , footer[ 80] ;         /* Discarded */
1720
1721      printf("Enter load file name ==> ");
1722      scanf(" %s", filename);
1723
1724      printf("Loading grid from <%s>.\n", filename);
1725      if (ReadMap3D(filename, grid, title, footer) == 0) {
1726        printf("load_grid: Unable to load grid from <%s>.\n", filename);
1727      }
1728  }
1729
1730  int load_grid_file(Map3D *grid,            /* Evidence grid */
1731                  char *filename)      /* Load file name */
1732  /*
1733    Load evidence grid from specified file
1734
```

```
1735        Returns 1 if successful, 0 otherwise
1736      */
1737    {
1738      char title[ 80] , footer[ 80] ;        /* Discarded */
1739
1740      printf("Loading grid from <%s>.\n", filename);
1741      if (ReadMap3D(filename, grid, title, footer) == 0) {
1742        printf("load_grid: Unable to load grid from <%s>.\n", filename);
1743        return (0);
1744      }
1745      return(1);
1746    }
1747
1748    int load_grid_file_com(Map3D *grid,        /* Evidence grid */
1749                           char *filename,     /* Load file name */
1750                           char *comment)      /* Comment string */
1751    /*
1752      Load evidence grid from specified file along with header comment
1753
1754      Returns 1 if successful, 0 otherwise
1755    */
1756    {
1757      char footer[ 80] ;                /* Discarded */
1758
1759      printf("Loading grid from <%s>.\n", filename);
1760      if (ReadMap3D(filename, grid, comment, footer) == 0) {
1761        printf("load_grid: Unable to load grid from <%s>.\n", filename);
1762        return (0);
1763      }
1764      return(1);
1765    }
1766
1767    void grid_count_occ(Map3D grid, int *occ, int *unocc)
1768    /*
1769      Count number of occupied and unoccupied cells in grid
1770    */
1771    {
1772      int x, y, z;                  /* Grid cell coordinates */
1773      int xsize, ysize, zsize;      /* Grid dimensions (#cells) */
1774      int p;                        /* Cell occupancy probability */
1775
1776      xsize = grid.msize[ 0] ;
1777      ysize = grid.msize[ 1] ;
1778      zsize = grid.msize[ 2] ;
1779
1780      *occ = 0;
1781      *unocc = 0;
1782
1783      for (x = 0; x < xsize; x++) {
1784        for (y = 0; y < ysize; y++) {
1785          for (z = 0; z < zsize; z++) {
1786          p = grid.mapm[ z * ysize * xsize + y * xsize + x] ;
1787          if (p > 0) {
1788            (*occ)++;
1789          }
1790          else if (p < 0) {
1791            (*unocc)++;
1792          }
```

1793   }
1794   }
1795   }
1796 }

APPENDIX E. FRONTIER-BASED EXPLORATION CODE – ROBOT.H

This appendix contains the header file for the routine that controls many of the robot's basic movement behaviors.

```
1    /*
2
3      robot.h
4
5      Header file for robot class for Nomad 200 Simulator.
6      Original code by Brian Yamauchi
7
8      Modifications for SCOUT THESIS
9      By Patrick A. Hillmeyer
10
11   */
12
13   #ifndef ROBOT_H
14
15   #define ROBOT_H
16
17   #include "Nclient.h"
18   #include "vector.h"
19   #include "misc.h"
20   #include "grid++.h"
21
22
23   // BEGIN SCOUT THESIS CHANGE
24
25   // These are the conversion macros from Nomadic that accept the steering
26   and
27   //  translation values as used for the Nomad 150 and 200 and calculate
28   the
29   //  differential-drive axis values for the Nomad Scout.
30
31   #define ROTATION_CONSTANT    0.118597  /* inches/degree (known to 100
32   ppm) */
33
34   #define RIGHT(trans, steer)  (trans +
35   (int)((float)steer*ROTATION_CONSTANT))
36   #define LEFT(trans, steer)   (trans -
37   (int)((float)steer*ROTATION_CONSTANT))
38
39   #define scout_vm(trans, steer)   vm(RIGHT(trans, steer), LEFT(trans,
40   steer), 0)
41   #define scout_pr(trans, steer)   pr(RIGHT(trans, steer), LEFT(trans,
42   steer), 0)
43
44   // END SCOUT THESIS CHANGE
45
46
47
48   const int NUM_SONAR = 16;    // Number of sensors of each type
49   const int NUM_IR = 16;       // Actually 0 for SCOUT but leave for now
50   const int NUM_RANGE = 16;
51
52   // BEGIN SCOUT THESIS CHANGE
```

```
53    const int NUM_TOUCH = 6;      // Scout only has 6 bumper swithes
54    // END SCOUT THESIS CHANGE
55
56    const int NUM_ARC = 16;        // Number of sensor arcs
57    const int ARC_SIZE = 3;        // Number of sensors in each arc
58    const int ARC_STEP = 1;        // Interval between first sensor
59                            // in each successive arc
60    const int ARC_OFFSET = -1;     // Value of first sensor of first arc (mod
61    16)
62
63    const int SONAR_ADDR = 17;     // State index for first sonar sensor .
64    const int IR_ADDR = 1;         // State index for first IR sensor
65    const int TOUCH_ADDR = 33;      // State index for touch sensors
66    const int LASER_MODE_ADDR = 42; // State index for laser mode
67
68    const int MAX_SONAR = 255;     // Maximum sonar reading
69
70    // BEGIN SCOUT THESIS CHANGE
71    const int MAX_IR = 0;          // Maximum IR reading - no IR on Scout
72    // END SCOUT THESIS CHANGE
73
74    const int MAX_RANGE = 255;     // Maximum range reading
75
76    const int SENSOR_SEP = 225;    // Separation between adjacent sensors
77                            // in degrees/10
78    // BEGIN SCOUT THESIS CHANGE
79    // this next setting for BUMPER_SEP can be left as is for now even
80    // though it is wrong for the Scout because the procedures that use
81    // it in agent.cc for recoiling from a bumper contact are not
82    implemented yet
83    // New NOTE - changed to 600 to fake out some code in robot.cc
84    // Needs a better fix though
85    const int BUMPER_SEP = 600;      // Separation between adjacent bumpers
86                            // in degrees/10
87
88
89    const int MAX_SPEED = 200;          // Maximum velocities
90    const int MAX_TURN_RATE = 300;  // From Nomadic .setup file for Scouts
91
92    const int MAX_ACCEL = 300;          // Maximum accelerations
93    const int MAX_TURN_ACCEL = 500;
94
95    const int DEFAULT_SPEED = 200;          // Default velocities
96    const int DEFAULT_TURN_RATE = 300;  // From Nomadic .setup file for
97    Scouts
98
99    // END SCOUT THESIS CHANGE
100
101   const int DEFAULT_ACCEL = 200;          // Default accelerations
102   const int DEFAULT_TURN_ACCEL = 500;
103
104   const int MOVE_TO_SPEED = 10;       // Speed for moving to (x,y)
105   const int FACE_TURN_RATE = 200;          // Turning rate for facing
106
107   const int MAX_CONT_TURN = 225;          // Maximum turn without
108   stopping
109   const int FACE_CONT_WAIT = 10;          // How long to wait for turn
110   to finish
```

```
111
112    const int ROBOT_MIN_SPEED = -200;         // Velocity limits (command)
113    const int ROBOT_MAX_SPEED = 200;
114    const int ROBOT_MIN_TURN = -100;
115    const int ROBOT_MAX_TURN = 100;
116
117    // BEGIN SCOUT THESIS CHANGE
118    // NOTE - changing no encoder parameters for now but might need to
119    //                   tweak them for the Scouts
120
121    // Dead reckoning parameters
122
123    const int ENCODER_COLOR = 19;         // Color of encoder graphic
124
125    // Minimum/maximum encoder rotation increment
126    //const double ENCODER_ROTATE_MIN = 1.0;
127    //const double ENCODER_ROTATE_MAX = 1.0;
128    const double ENCODER_ROTATE_MIN = 0.9;
129    const double ENCODER_ROTATE_MAX = 1.1;
130
131    // Encoder rotation bias
132    const double ENCODER_ROTATE_BIAS = 0.0;
133    //const double ENCODER_ROTATE_BIAS = 0.001;
134
135    // Minimum/maximum encoder translation increment
136    //const double ENCODER_TRANS_MIN = 1.0;
137    //const double ENCODER_TRANS_MAX = 1.0;
138    const double ENCODER_TRANS_MIN = 0.9;
139    const double ENCODER_TRANS_MAX = 1.1;
140
141    // Encoder translation bias
142    const double ENCODER_TRANS_BIAS = 0.0;
143    //const double ENCODER_TRANS_BIAS = 0.001;
144
145    // Cartesian move parameters
146
147    const int MOVE_XY_MAX_DIST = 1200;   // Maximum move (tenths inches)
148    const int MOVE_XY_MAX_ERROR = 1;     // Maximum move error (tenths
149    inches)
150
151    // END SCOUT THESIS CHANGE
152
153    // Building Axis Direction
154
155    const int AXIS = 2960;
156
157    // Arc directions
158
159    enum { FWD, FFL, FWD_LF, FLL, LF, BLL, BACK_LF, BBL,
160           BACK, BBR, BACK_RT, BRR, RT, FRR, FWD_RT, FFR };
161
162    // Timeout for movement commands
163
164    const unsigned char MOVE_TIMEOUT = 100;
165
166    // Robot class definition
167
168    class robot {
```

```
169    public:
170      int id;                          // Robot ID number
171      int x, y, theta, turret;         // Robot encoder position
172      int actual_x, actual_y, actual_theta;   // Robot actual position
173      double enc_x, enc_y, enc_theta;  // Accumulators for encoder position
174
175      int bumper_offset;               // Offset betwen base and bumpers
176
177      double trans_ctr;                // Total translation since
178    localization
179      double rot_ctr;                  // Total rotation since localization
180
181      int origin_x, origin_y;          // Room origin
182
183      int bumpers;                         // Bumper bit vector
184
185      vector sonar;                    // Sensor values
186      vector ir;
187      vector range;
188      vector touch;
189
190      vector abs_sonar;                // Absolute sensor values
191      vector abs_ir;
192      vector abs_range;
193      vector abs_touch;
194
195      vector arc;                      // Sensor arcs
196
197    /*
198      vector sonar(NUM_SONAR);         // Sensor values
199      vector ir(NUM_IR);
200      vector range(NUM_RANGE);
201      vector touch(NUM_TOUCH);
202
203      vector abs_sonar(NUM_SONAR);            // Absolute sensor values
204      vector abs_ir(NUM_IR);
205      vector abs_range(NUM_RANGE);
206      vector abs_touch(NUM_TOUCH);
207
208      vector arc(NUM_ARC);          // Sensor arcs
209    */
210
211      robot(void);                  // Constructor
212
213      void update(void);           // Sensor update
214
215      void set_default_velocity(void);  // Set default trans/base/turret
216    speed
217
218      void maint_err(void);        // Maintain encoder error at new position
219
220      // Relative move of <speed>/10 inches forward while turning both
221    turret
222      // and base by <angle>/10 degrees (+ = ccw, - = cw)
223
224      void move(int speed, int angle);
225
226      // Relative move of <speed>/10 inches forward
```

174

```
227
228     void fwd(int speed);
229
230     // Turn base by <angle>/10 degrees (+ = ccw, - = cw)
231
232     void turn(int angle);
233
234     // Set origin to current position
235
236     void set_origin_here(void);
237
238     // Set origin to (o_x, o_y)
239
240     void set_origin_loc(int o_x, int o_y);
241
242     // Convert angle to sensor index
243
244     int theta2sensor(double theta);
245
246     // Convert sensor index to angle
247
248     double sensor2theta(int sensor);
249
250     // Return 1 if all range sensors in an arc <width> x 2 sensors wide
251     // centered around sensor <ctr> return greater or equal to <dist>,
252     // 0 otherwise.
253
254     int check_clear(int ctr, int width, int dist);
255
256     // Wrap index to [0..NUM_SENSORS]
257
258     int sensor_wrap(int index);
259
260     // Turn off sensors
261
262     void shutdown(void);
263
264     // Move robot to <x, y> (world coords, tenths of inch)
265
266     int move_to_xy(int cx, int cy);
267
268     // Turn robot to face <angle> accurately
269
270     void face_angle(int angle);
271
272     // Turn robot to face <angle> quickly
273
274     void face_angle_fast(int angle);
275
276     // Turn robot to face <angle> quickly, without stopping
277
278     void face_angle_cont(int angle);
279
280     // Align turret with base
281
282     void turret_align(void);
283
284     // Relative Cartesian move to <x, y> (world coords, tenths of inches)
```

```
285
286       void move_rel(int x, int y);
287
288       // Sensor functions
289
290       void sonar_on(void);
291       void sonar_single(int index);            // Index of sensor to fire
292       void sonar_off(void);
293       void ir_on(void);
294       void ir_single(int index);          // Index of sensor to fire
295       void ir_off(void);
296       void laser_on(void);
297       void laser_off(void);
298
299       // Wait for the robot to start moving (any motor)
300
301       void wait_start(void);
302
303    private:
304
305       // Initialization functions
306
307       void initialize_sensors(void);
308
309       // Update functions
310
311       void update_dr(void);
312       void update_range_arcs(void);
313       void update_arc(int &av, int first, int last);
314
315       // Cleanup functions
316       void deactivate_sonar(void);
317
318
319    };
320
321    #endif
```

APPENDIX F. FRONTIER-BASED EXPLORATION CODE – ROBOT.CC

This appendix contains the source code for the routine that controls many of the robot's basic movement behaviors.

```
1    /*
2
3       robot.cc
4
5       Robot class for Nomad 200 Simulator.
6       Original code by Brian Yamauchi
7
8       Modifications for SCOUT THESIS
9       By Patrick A. Hillmeyer
10
11   */
12
13   #include <iostream.h>
14   #include <math.h>
15
16   #include "robot.h"
17   #include "drand.h"
18   #include "irand.h"
19
20   // Dead reckoning mode (actual, independent, or error)
21
22   #define DR_ACTUAL
23
24   // Touch vector mask
25
26   const int touch_mask[ 20] = { 1, 2, 4, 8, 16,
27                                 32, 64, 128, 256, 512,
28                                 1024, 2048, 4096, 8192, 16384,
29                                 32768, 65536, 131072, 262144, 524288 };
30
31   // Forward contact mask
32
33   const int FWD_CONTACT = 1015839;
34
35   robot::robot(void):sonar(NUM_SONAR), ir(NUM_IR), range(NUM_RANGE),
36                      touch(NUM_TOUCH), abs_sonar(NUM_SONAR),
37   abs_ir(NUM_IR),
38                      abs_range(NUM_RANGE), abs_touch(NUM_TOUCH),
39   arc(NUM_ARC)
40   {
41      int rx, ry, rtheta;          // Robot home position (1/10 inch, 1/10
42   degree)
43
44      // Connect to server and activate all sensors
45
46      cout << "Enter Nserver host name ==> ";
47      cin >> SERVER_MACHINE_NAME;
48
49      cout << "Enter Nserver robot ID ==> ";
50      cin >> id;
51
52      connect_robot(id);
```

```
53    initialize_sensors();
54    set_default_velocity();
55
56    // Initialize origin
57
58    origin_x = 0;
59    origin_y = 0;
60
61    // Initialize translation/rotation counters
62
63    trans_ctr = 0.0;
64    rot_ctr = 0.0;
65
66    // Zero the robot
67
68    //  zr();
69
70    // Set robot initial position
71
72    //  tk("Align me.");
73
74    cout << "Enter robot x y theta (no commas) ==> ";
75    cin >> rx >> ry >> rtheta;
76
77    gs();
78    bumper_offset = State[ 36]  - rtheta;
79
80    place_robot(rx, ry, rtheta, rtheta);
81
82    // Initialize actual position
83
84    gs();
85    actual_x = State[ 34] ;
86    actual_y = State[ 35] ;
87    actual_theta = State[ 36] ;    // DR heading
88 //  actual_theta = 3600 - wrap(State[ 43]  - AXIS, 0, 3600);  // Compass
89 heading
90
91    // Initialize encoder accumulators
92
93    enc_x = (double) actual_x;
94    enc_y = (double) actual_y;
95    enc_theta = (double) actual_theta;
96
97    // Initialize estimated position
98
99    x = round(enc_x);
100   y = round(enc_y);
101   theta = round(enc_theta);
102
103   // Display robot estimated position
104
105 //  draw_robot(x, y, theta, theta, ENCODER_COLOR);
106
107   // Updater robot state
108
109   update();
110 }
```

```
111
112    void robot::maint_err(void)
113    {
114        // Maintain encoder error at new position
115
116        int ex, ey, etheta;
117
118        ex = x - actual_x;
119        ey = y - actual_y;
120        etheta = theta - actual_theta;
121
122        gs();
123        place_robot(State[ 34] , State[ 35] , State[ 36] , State[ 37] );
124
125        actual_x = State[ 34] ;
126        actual_y = State[ 35] ;
127        actual_theta = State[ 36] ;
128
129        x = actual_x + ex;
130        y = actual_y + ey;
131        theta = actual_theta + etheta;
132    }
133
134    void robot::set_default_velocity()
135    {
136      sp(DEFAULT_SPEED, DEFAULT_TURN_RATE, 0);   // TEMP FIX for SCOUT
137      ac(DEFAULT_ACCEL, DEFAULT_TURN_ACCEL, 0); // TEMP FIX for SCOUT
138    }
139
140    void robot::update(void)
141    {
142      // Update values for position <x, y, theta>, sonar <sonar>, infrared
143      // sensors <ir>.  Also update range arcs.
144
145      int range_offset;              // Rotation offset for range sensors
146      int touch_offset;              // Rotation offset for touch sensors
147      int i;
148
149      gs();
150
151      update_dr();
152
153      range_offset = (int) ((double) theta / (double) SENSOR_SEP + 0.5);
154
155    // NOTE - need to fix this BUMPER_SEP dependency later for SCOUT
156
157      touch_offset = wrap((int) ((double) (theta + bumper_offset)
158                          / (double) BUMPER_SEP + 0.5),
159                    NUM_TOUCH);
160
161      //  cout << "Offset = " << range_offset << "" << endl;
162
163      for (i = 0; i < NUM_SONAR; i++) {
164        sonar[ i] = State[ i + SONAR_ADDR] ;
165        abs_sonar[ i] = State[ wrap(i - range_offset, NUM_SONAR) +
166    SONAR_ADDR] ;
167
```

```
168        //      cout << "i = " << i << " : range_offset i = " << wrap(i -
169   range_offset, NUM_SONAR) <<
170        //         " : sonar[" << i << "] = " << sonar[ i] << "" << endl;
171     }
172
173   // BEGIN SCOUT THESIS CHANGE
174   // Comment out IR code and only depend on sonars
175
176   // SCOUT THESIS CHANGE -  correct error where sensor updates below were
177   left out of loop
178     for (i = 0; i < NUM_SONAR; i++) {
179       abs_range[ i] = abs_sonar [ i] ;
180       range[ i] = sonar[ i] ;
181   // cout << "Just set by sonar[ i] value : range[" << i << "] = " <<
182   range[ i]
183   //           << endl;          // TEMP FIX
184   // cout << "Just set by abs_sonar[ i] value : abs_range[" << i << "] = "
185   //           << abs_range[ i] << endl;       // TEMP FIX
186
187     }   // end for loop
188
189   //   for (i = 0; i < NUM_IR; i++) {
190   //     ir[ i] = State[ i + IR_ADDR] ;
191   //     abs_ir[ i] = State[ wrap(i - range_offset, NUM_IR) + IR_ADDR] ;
192
193        //      cout << "i = " << i << " : range_offset i = " << wrap(i -
194   range_offset, NUM_IR) <<
195        //         " : ir[" << i << "] = " << ir[ i] << "" << endl;
196   //   }
197
198   //   for (i = 0; i < NUM_RANGE; i++) {
199   //     if (abs_ir[ i] < abs_sonar[ i] ) {
200   //        abs_range[ i] = abs_ir[ i] ;
201   //     }
202   //     else {
203   //        abs_range[ i] = abs_sonar[ i] ;
204   //     }
205   //
206   //     if (ir[ i] < sonar[ i] ) {
207   //        range[ i] = ir[ i] ;
208   //     }
209   //     else {
210   //        range[ i] = sonar[ i] ;
211   //     }
212   //   }
213
214   // END SCOUT THESIS CHANGE
215
216     for (i = 0; i < NUM_RANGE; i++) {
217       if (range[ i] > MAX_RANGE) {
218         range[ i] = MAX_RANGE;
219   // cout << "Compared against MAX_RANGE : range[" << i << "] = " <<
220   range[ i]
221   //           << endl;       // TEMP FIX
222       }
223       if (abs_range[ i] > MAX_RANGE) {
224         abs_range[ i] = MAX_RANGE;
```

```cpp
225    // cout << "Compared against MAX_RANGE : abs_range[" << i << "] = " <<
226    abs_range[ i]
227    //         << endl;     // TEMP FIX
228        }
229      }     // end for loop
230
231      update_range_arcs();
232
233      bumpers = State[ TOUCH_ADDR] ;
234      //   if (bumpers != 0) {
235      //       cout << "Bumper state = " << bumpers << "" << endl;
236      //   }
237
238    // NOTE - touch_offset depends on BUMPER_SET - needs fix for SCOUT
239
240      for (i = 0; i < NUM_TOUCH; i++) {
241        if (bumpers &
242          touch_mask[ wrap(i + touch_offset, NUM_TOUCH)] ) {
243          touch[ i] = 1;
244          cout << "Contact on bumper " << i << " (abs index = " <<
245          wrap(i + touch_offset, NUM_TOUCH) << ")" << endl;
246        }
247        else {
248          touch[ i] = 0;
249        }
250      }
251
252    // cout << "Sonar = " << sonar << endl;     // TEMP FIX
253      //   cout << "IR = " << ir << endl;
254    // cout << "Range = " << range << endl;     // TEMP FIX
255    // cout << "Arcs = " << arc << endl;        // TEMP FIX
256      //   cout << "Touch = " << touch << endl;
257    }
258
259    void robot::update_dr(void)
260    {
261      // Update dead reckoning
262
263      double dx, dy, dtheta;            // Motion since last update
264      int dtheta_mag;           // Magnitude of rotation
265      int dtheta_sgn;           // Direction of rotation (+ ccw, - cw)
266
267      double vec_r, vec_theta;          // Motion vector
268      double inc;                       // Motion increment
269      double ctheta, stheta;            // Components along x-axis and y-
270    axis
271      double trans_step;                // Length of current translation
272    step
273
274      int i;
275
276      dx = (double) (State[ 34] - actual_x);
277      dy = (double) (State[ 35] - actual_y);
278      dtheta = angle_sgn_diff((double) State[ 36] / 10.0,
279                      (double) actual_theta / 10.0) * 10.0;
280
281      actual_x = State[ 34] ;
282      actual_y = State[ 35] ;
```

```
283        actual_theta = State[ 36] ;    // DR heading
284   //  actual_theta = 3600 - wrap(State[ 43]  - AXIS, 0, 3600);   // Compass
285   heading
286
287   #ifdef DR_ACTUAL
288
289      // Dead reckoning always returns actual position
290
291      x = actual_x;
292      y = actual_y;
293      theta = actual_theta;
294   · // SCOUT THESIS CHANGE - comment out original turret line
295      // set turret to be same as SCOUT heading angle
296      //   turret = State[ 37] ;
297        turret = theta;
298
299   #endif // DR_ACTUAL
300
301   #ifdef DR_INDEP
302
303      // Dead reckoning is updated by actual displacements, but may be set
304      // independently
305
306      draw_robot(x, y, theta, theta, ENCODER_COLOR);
307
308      x += (int) dx;
309      y += (int) dy;
310      theta += (int) dtheta;
311
312      draw_robot(x, y, theta, theta, ENCODER_COLOR);
313
314   #endif // DR_INDEP
315
316   #ifdef DR_ERROR
317
318      // Dead reckoning accumulates error over time
319
320      draw_robot(x, y, theta, theta, ENCODER_COLOR);
321
322      rot_ctr += fabs(dtheta);
323
324      dtheta_mag = (int) fabs(dtheta);
325      dtheta_sgn = sgn(dtheta);
326      for (i = 0; i < dtheta_mag; i++) {
327        enc_theta += (double) dtheta_sgn *
328          rdrand(ENCODER_ROTATE_MIN, ENCODER_ROTATE_MAX) +
329   ENCODER_ROTATE_BIAS;
330      }
331      enc_theta = angle_wrap(enc_theta / 10.0) * 10.0;
332
333      theta = round(enc_theta);
334
335      vec_r = hypot(dx, dy);
336
337      if (vec_r > 0.0) {
338        trans_ctr += vec_r;
339
340        vec_theta = (double) theta / 10.0;
```

```
341        if (angle_diff(vec_theta, atan2(dy, dx) * RAD2DEG) > 90.0) {
342          vec_theta = angle_wrap(vec_theta + 180.0);
343        }
344
345        ctheta = cos(vec_theta * DEG2RAD);
346        stheta = sin(vec_theta * DEG2RAD);
347
348        for (i = 0; i < (int) vec_r; i++) {
349          trans_step = rdrand(ENCODER_TRANS_MIN, ENCODER_TRANS_MAX) +
350          ENCODER_TRANS_BIAS;
351          trans_step = 1.0;
352          enc_x += ctheta * trans_step;
353          enc_y += stheta * trans_step;
354        }
355
356        enc_x += ctheta * (vec_r - (int) vec_r);
357        enc_y += stheta * (vec_r - (int) vec_r);
358
359        x = round(enc_x);
360        y = round(enc_y);
361      }
362
363    draw_robot(x, y, theta, theta, ENCODER_COLOR);
364
365  /*  cout << "Actual: (" << actual_x << ", " << actual_y << ") <" <<
366  actual_theta
367      << "> -- Encoder: (" << enc_x << ", " << enc_y << ") <" << enc_theta
368  <<
369        "> -- Error: (" << enc_x - (double) actual_x << ", " <<
370        enc_y - (double) actual_y << ") <" <<
371          round(angle_sgn_diff(enc_theta / 10.0,
372                              (double) actual_theta / 10.0)
373            * 10.0) << ">" << endl;
374
375    cout << "Total: translation = " << trans_ctr << " : rotation = " <<
376      rot_ctr << endl;*/
377
378  #endif // DR_ERROR
379
380    return;
381  }
382
383  void robot::update_range_arcs(void)
384  {
385    // Update range arcs.  The value of the arc is equal to the minimum
386    // range reading of a sensor that is included in that arc.
387
388    int i, first, last;
389
390    for (i = 0; i < NUM_ARC; i++) {
391      first = wrap(i * ARC_STEP + ARC_OFFSET, NUM_RANGE);
392      last = wrap(first + ARC_SIZE - 1, NUM_RANGE);
393
394      arc[i] = range.min(first, last);
395    }
396  }
397
398  void robot::sonar_on(void)
```

```
399    {
400       // Activate all sonar sensors
401
402          int sn_order[ 16] ;     // Sonar firing order
403
404          /* set firing rate and sequence of all sonar */
405          sn_order[ 0]  =   0; sn_order[ 1]  = 10; sn_order[ 2]  =   6;
406          sn_order[ 3]  = 14; sn_order[ 4]  =   2; sn_order[ 5]  = 12;
407          sn_order[ 6]  =   4; sn_order[ 7]  =   9; sn_order[ 8]  =   1;
408          sn_order[ 9]  = 13; sn_order[ 10]  =   5; sn_order[ 11]  = 15;
409          sn_order[ 12]  = 7; sn_order[ 13]  =   11; sn_order[ 14]  = 3;
410          sn_order[ 15]  = 8;
411          conf_sn (10, sn_order);      // TEMP FIX SET LONGER SONAR FIRING TIME
412    }
413
414    void robot::sonar_single(int index) // Index of sensor to fire
415    {
416       // Activate one sonar sensor
417
418       int sn_order[ 16] ;      // Sonar firing order
419
420       sn_order[ 0]  = index;
421       sn_order[ 1]  = 255;
422
423       conf_sn (12, sn_order);
424    }
425
426    void robot::sonar_off(void)
427    {
428       // Deactivate all sonar sensors
429
430       int sn_order[ 16] ;      // Sonar firing order
431
432       sn_order[ 0]  = 255;
433       conf_sn(1, sn_order);
434    }
435
436
437    // BEGIN SCOUT THESIS CHANGE
438    // let the IRs and laser be configured - just comment out the activation
439    //  in the following procedures
440
441    void robot::ir_on(void)
442    {
443       // Activate all IR sensors
444
445          int ir_order[ 16] ;    // IR firing order
446
447       /* set firing rate and sequence of all IR */
448          ir_order[ 0]  =   0; ir_order[ 1]  = 10; ir_order[ 2]  =   6;
449          ir_order[ 3]  = 14; ir_order[ 4]  =   2; ir_order[ 5]  = 12;
450          ir_order[ 6]  =   4; ir_order[ 7]  =   9; ir_order[ 8]  =   1;
451          ir_order[ 9]  = 13; ir_order[ 10]  =   5; ir_order[ 11]  = 15;
452          ir_order[ 12]  = 7; ir_order[ 13]  =   11; ir_order[ 14]  = 3;
453          ir_order[ 15]  = 8;
454    //    conf_ir (2, ir_order);
455    }
456
```

```cpp
457   void robot::ir_single(int index)     // Index of sensor to fire
458   {
459      // Activate one IR sensor
460
461      int ir_order[16];      // IR firing order
462
463      ir_order[0] = index;
464      ir_order[1] = 255;
465
466   //  conf_ir(2, ir_order);
467   }
468
469   void robot::ir_off(void)
470   {
471      // Deactivate all IR sensors
472
473      int ir_order[16];      // IR firing order
474
475      ir_order[0] = 255;
476   //  conf_ir(2, ir_order);
477   }
478
479   void robot::laser_on(void)
480   {
481      // Activate laser
482
483   //  conf_ls(LASER_MODE_ON, THRESHHOLD, WIDTH, NUMDATA, AVG);
484   }
485
486   void robot::laser_off(void)
487   {
488      // Deactivate laser
489
490   //  conf_ls(LASER_MODE_OFF, THRESHHOLD, WIDTH, NUMDATA, AVG);
491   }
492
493   void robot::initialize_sensors(void)
494   {
495      /*
496        Activate all robot sensors
497        */
498   //    static int ir_on[16] ={0, 10, 6, 14, 2, 12, 4, 9, 1, 13, 5, 15, 7,
499   11, 3, 8};
500   //    static int ir_off[16] ={255, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
501   13, 14, 15};
502
503      static int sn_on[16] ={0, 10, 6, 14, 2, 12, 4, 9, 1, 13, 5, 15, 7,
504   11, 3, 8};
505      static int sn_off[16] ={255, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
506   13, 14, 15};
507
508      int i;
509
510      /*  init_sensors(); */
511
512   //    conf_ir(0, ir_on);
513      conf_sn(10, sn_on);   //TEMP FIX - set longer slower firing time
514
```

```
515      Smask[ 42] = 1;
516   //    conf_ls(LASER_MODE_ON, THRESHHOLD, WIDTH, NUMDATA, AVG);
517
518      /*  conf_cp(1); */  /* =-=-= doesn't work with Ndirect.o */
519
520      posDataRequest(POS_SONAR);          // Just get the sonar data for
521   the SCOUT
522      if (posDataCheck() != (POS_SONAR)) {        // Just check for the
523   sonar data for the SCOUT
524         cout << "\nERROR: Could not set up pose info for sensors.\n";
525         exit(-1);
526      }
527   }
528
529   // END SCOUT THESIS CHANGE
530
531
532
533   void robot::move(int speed, int angle)
534   {
535      // Relative move of <speed>/10 inches forward while turning
536      // base and turret by <angle>/10 degrees (+ = ccw, - = cw)
537
538      int vel_speed, vel_angle;    // Velocity commands
539
540      vel_speed = limit(speed, ROBOT_MIN_SPEED, ROBOT_MAX_SPEED);
541      vel_angle = limit(angle, ROBOT_MIN_TURN, ROBOT_MAX_TURN);
542
543   // BEGIN SCOUT THESIS CHANGE
544      scout_vm(vel_speed, vel_angle);
545
546   //   scout_pr(speed, angle);
547   }
548
549   void robot::fwd(int speed)
550   {
551      // Relative move of <speed>/10 inches forward
552
553   // TEMP SCOUT FIX- change pr cmds to vm cmds and comment out the wait
554      scout_vm(speed, 0);
555   //  ws(1, 1, 0, 5);      TEMP FIX - comment out the wait
556   }
557
558   void robot::turn(int angle)
559   {
560      // Turn base and turret by <angle>/10 degrees (+ = ccw, - = cw)
561
562   // TEMP FIX - change pr cmds to vm cmds and comment out the wait
563      scout_vm(0, angle);
564   //    ws(1, 1, 0, 5);  TEMP FIX - comment out the wait
565   }
566
567   // END SCOUT THESIS CHANGE
568
569
570   void robot::set_origin_here(void)
571   {
572      // Define current position and orientation as origin
```

```
573
574    /*   dp(0, 0);
575      da(0, 0);*/
576
577      origin_x = State[ 34] ;
578      origin_y = State[ 35] ;
579
580      x = 0;
581      y = 0;
582      theta = 0;
583
584      cout << "Setting origin to (" << origin_x << ", " << origin_y << ")."
585   << endl;
586   }
587
588   void robot::set_origin_loc(int o_x, int o_y)
589   {
590      // Define new origin relative to current origin
591
592      origin_x += o_x;
593      origin_y += o_y;
594
595      x = 0;
596      y = 0;
597      theta = 0;
598
599      cout << "Setting origin to (" << origin_x << ", " << origin_y << ")."
600   << endl;
601   }
602
603   int robot::theta2sensor(double theta)
604   {
605       // Convert angle to sensor index
606
607       int sensor;
608
609       sensor = (int) (theta * 10.0) / SENSOR_SEP;
610       return(sensor);
611   }
612
613   double robot::sensor2theta(int sensor)
614   {
615       // Convert sensor index to angle
616
617       double theta;
618
619       theta = (double) (sensor * SENSOR_SEP) / 10.0;
620       return(theta);
621   }
622
623   int robot::sensor_wrap(int index)
624   {
625       // Wrap index to [ 0..NUM_RANGE]
626
627       int windex;
628
629       windex = wrap(index, NUM_RANGE);
630       return(windex);
```

```
631   }
632
633   int robot::check_clear(int ctr, int width, int dist)
634   {
635       // Return 1 if all range sensors in an arc <width> x 2 sensors wide
636       // centered around sensor <ctr> return greater or equal to <dist>,
637       // 0 otherwise.
638
639       int left, right;
640       int min_dist;
641
642       left = sensor_wrap(ctr + width);
643       right = sensor_wrap(ctr - width);
644
645       min_dist = range.min(right, left);
646
647       if (min_dist >= dist) {
648           return(1);
649       }
650       else {
651           return(0);
652       }
653   }
654
655   void robot::shutdown(void)
656   {
657     sonar_off();
658     ir_off();
659     laser_off();
660   }
661
662   int robot::move_to_xy(int cx, int cy)
663   {
664       // Move robot to <x, y> (world coords, tenths of inch)
665
666       int dx, dy;          // Difference between current and desired
667   positions
668       double dist;  // Distance to destination
669       double angle; // Bearing to destination
670       double mturn; // Turn command
671
672   // BEGIN SCOUT THESIS CHANGE
673       scout_vm(0, 0);
674   // END SCOUT THESIS CHANGE
675       sp(MOVE_TO_SPEED, DEFAULT_TURN_RATE, 0);  // TEMP FIX for SCOUT
676
677       update();
678
679       cout << "current position (" << x << ", " << y << ") : destination
680   ("
681           << cx << ", " << cy << ")" << endl;
682
683       dx = cx - x;
684       dy = cy - y;
685
686       dist = hypot((double) dx, (double) dy);
687       if (dist == 0.0) {
688           angle = 0.0;
```

188

```
689          }
690          else {
691            angle = atan2((double) dy, (double) dx) * RAD2DEG;
692          }
693
694          cout << "distance = " << dist << " : angle = " << angle << endl;
695
696          while((dist > MOVE_XY_MAX_ERROR) && (touch.max(0, NUM_TOUCH - 1) ==
697    0)) {
698            if (dist > MOVE_XY_MAX_DIST) {
699                cout << "Destination too far (" << dist / 10.0 << " inches)"
700    <<
701                    endl;
702                return(0);
703            }
704
705            mturn = (int) (angle_sgn_diff(angle, (double) theta / 10.0) *
706    10.0);
707    // BEGIN SCOUT THESIS CHANGE
708
709    // TEMP FIX - change pr cmds to vm cmds and commnent out the ws cmds
710            scout_vm(0, (int) mturn);
711    //     ws(1, 1, 0, 100);      TEMP FIX - comment out wait
712
713            scout_vm((int) dist, 0);
714    //     ws(1, 1, 0, 100);      TEMP FIX - comment out wait
715
716    // END SCOUT THESIS CHANGE
717
718            update();
719
720            cout << "current position (" << x << ", " << y << ") : destination
721    ("
722                << cx << ", " << cy << ")" << endl;
723
724            dx = cx - x;
725            dy = cy - y;
726
727            dist = hypot((double) dx, (double) dy);
728            if (dist == 0.0) {
729                angle = 0.0;
730            }
731            else {
732                angle = atan2((double) dy, (double) dx) * RAD2DEG;
733            }
734
735            cout << "distance = " << dist << " : angle = " << angle << endl;
736        }
737
738        set_default_velocity();
739
740        return(1);
741    }
742
743    void robot::face_angle(int angle)    // Desired angle (1/10 degree)
744    {
745        // Turn robot to face <angle> accurately
746
```

```
747    int dtheta;                // Difference between current and desired angle
748
749    cout << "Facing angle <" << angle << ">" << endl;
750
751 // BEGIN SCOUT THESIS CHANGE
752
753    scout_vm(0, 0);
754    sp(DEFAULT_SPEED, FACE_TURN_RATE, 0);   // TEMP FIX for SCOUT
755
756    update();
757    dtheta = (int)
758      (angle_sgn_diff((double) angle / 10.0, (double) theta / 10.0) *
759 10.0);
760
761    while(dtheta != 0) {
762      cout << "current angle = " << theta << " : turn = " << dtheta <<
763 endl;
764
765 // TEMP FIX - change below to vm vice pr and comment out the wait
766      scout_vm(0, dtheta);
767 //    ws(1, 1, 0, 100);    TEMP FIX comment out the wait
768
769 // END SCOUT THESIS CHANGE
770
771      update();
772      dtheta = (int)
773        (angle_sgn_diff((double) angle / 10.0, (double) theta / 10.0) *
774 10.0);
775    }
776
777    cout << "Alignment complete." << endl;
778
779    set_default_velocity();
780 }
781
782 void robot::face_angle_fast(int angle)     // Desired angle (1/10 degree)
783 {
784    // Turn robot to face <angle> quickly
785
786    int dtheta;                // Difference between current and desired angle
787
788    dtheta = (int)
789      (angle_sgn_diff((double) angle / 10.0, (double) theta / 10.0) *
790 10.0);
791
792 // TEMP FIX for SCOUT to line below
793 //   cout << "face_angle_fast(" << angle << ") : scout_vm(0, " << dtheta
794 //        << ")" << endl;    // TEMP FIX for SCOUT
795
796 // BEGIN SCOUT THESIS CHANGE
797
798 // TEMP FIX - change pr cmds to vm cmds and comment out the wait
799    scout_vm(0, dtheta);
800 //  ws(1, 1, 0, 10);    TEMP FIX - comment out the wait
801 // END SCOUT THESIS CHANGE
802 }
803
804 void robot::face_angle_cont(int angle)     // Desired angle (1/10 degree)
```

190

```
805    {
806       // Turn robot to face <angle> quickly, without stopping
807
808       int dtheta;                // Difference between current and desired angle
809
810       dtheta = (int)
811          (angle_sgn_diff((double) angle / 10.0, (double) theta / 10.0) *
812       10.0);
813
814       cout << "face_angle_cont(" << angle << ") : pr(0, " << dtheta << ", "
815             << dtheta << ")" << endl;
816
817       if ((dtheta < -MAX_CONT_TURN) || (dtheta > MAX_CONT_TURN)) {
818    //     mv(MV_VM, 0, MV_PR, dtheta, MV_PR, dtheta);  // TEMP FIX - comment
819    this line out
820          mv(MV_VM, 0, MV_PR, dtheta, MV_IGNORE, 0);  // TEMP FIX - try to fix
821    SCOUT problem
822          ws(1, 1, 0, FACE_CONT_WAIT);    // wait for both wheels to stop for
823    SCOUT
824       }
825       else {
826    //     mv(MV_IGNORE, 0, MV_PR, dtheta, MV_PR, dtheta);  // TEMP FIX -
827    comment this line out
828          mv(MV_IGNORE, 0, MV_PR, dtheta, MV_IGNORE, 0);   // TEMP FIX - try
829    to fix SCOUT problem
830       }
831    }
832
833    // BEGIN SCOUT THESIS CHANGE
834    // do not need this next routine because there is no separate
835    // turret to align on the SCOUT
836    // leave as is because call to turret_align has been commented out
837    // in agent.cc
838    void robot::turret_align(void)
839    {
840        // Align turret with base
841
842        int turret;         // Turret angle
843        int dtheta;         // Difference between turret and base
844
845        scout_vm(0, 0);
846        sp(DEFAULT_SPEED, FACE_TURN_RATE, 0);    // TEMP FIX for SCOUT
847
848        update();
849        turret = State[ 37] ;
850        dtheta = 0;      // fake it for SCOUT
851    //    dtheta = wrap(actual_theta - turret, -1800, 1800);
852
853        while(dtheta != 0) {
854           scout_vm(0, 0);    // TEMP TECK for SCOUT
855           ws(0, 0, 0, 100);
856
857           update();
858           turret = State[ 37] ;
859           dtheta = wrap(actual_theta - turret, -1800, 1800);
860        }
861
862    // END SCOUT THESIS CHANGE
```

```
863          set_default_velocity();
864    }
865
866    void robot::move_rel(int x, int y)
867    {
868          // Relative Cartesian move to <x, y> (world coords, tenths of
869    inches)
870
871          int old_angle;
872          int move_angle;
873          int move_dist;
874
875          update();
876          old_angle = State[ 36] ;
877
878          move_angle = (int) (atan2((double) y, (double) x) * RAD2DEG * 10.0);
879          move_dist = (int) hypot((double) x, (double) y);
880
881          face_angle(move_angle);
882
883          sp(MOVE_TO_SPEED, DEFAULT_TURN_RATE, 0);    // TEMP FIX for SCOUT
884    // BEGIN SCOUT THESIS CHANGE
885
886    // TEMP FIX - change pr to vm and comment out the wait
887          scout_vm(move_dist, 0);
888    //    ws(1, 1, 0, 100);    TEMP FIX - comment out the wait
889    // END SCOUT THESIS CHANGE
890          set_default_velocity();
891
892          face_angle(old_angle);
893    }
894
895    void robot::wait_start(void)
896    {
897      // Wait for the robot to start moving (any motor)
898
899      gs();
900
901      while((State[ STATE_VEL_TRANS] == 0) &&
902           (State[ STATE_VEL_STEER] == 0) &&
903           (State[ STATE_VEL_TURRET] == 0)) {
904        gs();
905      }
906    }
```

APPENDIX G. FRONTIER-BASED EXPLORATION CODE – AGENT.H

This appendix contains the header file for the routine that controls the robot's exploration behaviors.

```
1    /*
2
3      agent.h
4
5      Header file for agent class
6
7    */
8
9    #ifndef AGENT_H
10   #define AGENT_H
11
12   #include "drand.h"
13   #include "irand.h"
14   #include "robot.h"
15   #include "place_net.h"
16   #include "arb.h"
17   #include "control_panel.h"
18   #include "bar_graph.h"
19   #include "mobstacle.h"
20   #include "comm++.h"
21   #include "comm.h"
22   #include "frontier.h"
23   #include "path.h"
24
25   // BEGIN SCOUT THESIS CHANGE
26
27   // These are the conversion macros from Nomadic that accept the steering
28   and
29   //  translation values as used for the Nomad 150 and 200 and calculate
30   the
31   //  differential-drive axis values for the Nomad Scout.
32
33   #define ROTATION_CONSTANT    0.118597  /* inches/degree (known to 100
34   ppm) */
35
36   #define RIGHT(trans, steer)  (trans +
37   (int)((float)steer*ROTATION_CONSTANT))
38   #define LEFT(trans, steer)   (trans -
39   (int)((float)steer*ROTATION_CONSTANT))
40
41   #define scout_vm(trans, steer)   vm(RIGHT(trans, steer), LEFT(trans,
42   steer), 0)
43   #define scout_pr(trans, steer)   pr(RIGHT(trans, steer), LEFT(trans,
44   steer), 0)
45
46   // END SCOUT THESIS CHANGE
47
48
49
50   // Motor control parameters
51
52   const int SPEED_RES = 40;
```

```
53    const double SPEED_MIN = -100.0;
54    const double SPEED_MAX = 100.0;
55    const double SPEED_DEF = 0.0;
56    const double SPEED_NOISE = 5.0;
57
58    const int TURN_RES = 32;
59    const double TURN_MIN = -180.0;                    .
60    const double TURN_MAX = 180.0;
61    const double TURN_DEF = 0.0;
62    const double TURN_NOISE = 5.0;
63
64    // Random turn to escape stasis
65
66    const double RAND_TURN = 10.0;
67
68    // Speed arbitrator window parameters
69
70    const int SPWIN_X = 570;              // x-coord of top
71    const int SPWIN_Y = 460;              // y-coord of left side
72    const int SPWIN_WIDTH = 175;          // Window width
73    const int SPWIN_HEIGHT = 50;          // Window height
74    const double SPWIN_MIN = -20.0;            // Minimum vote total
75    const double SPWIN_MAX = 20.0;             // Maximum vote total
76
77    // Turn arbitrator window parameters
78
79    const int TUWIN_X = 570;              // x-coord of top
80    const int TUWIN_Y = 540;              // y-coord of left side
81    const int TUWIN_WIDTH = 175;          // Window width
82    const int TUWIN_HEIGHT = 50;          // Window height
83    const double TUWIN_MIN = -20.0;            // Minimum vote total
84    const double TUWIN_MAX = 20.0;             // Maximum vote total
85
86    // Power constants
87
88    const double CPU_FULL_VOLTAGE = 12.0;
89    const double CPU_DANGER_VOLTAGE = 11.0;
90
91    const double MOTOR_FULL_VOLTAGE = 12.0;
92    const double MOTOR_DANGER_VOLTAGE = 11.0;
93
94
95    /********** BEHAVIOR CONSTANTS **********/
96
97    // Bump halt
98
99    const int BUMP_SLEEP = 10;     // Number of seconds to sleep
100
101   // Recoil
102
103   const double RECOIL_SPEED = 100.0;
104   const double RECOIL_SPEED_SIGMA = 25.0;
105   const double RECOIL_TURN = 45.0;
106   const double RECOIL_TURN_SIGMA = 10.0;
107   const double RECOIL_WT = 10.0;
108
109   // Maintain alignment
110
```

```cpp
111    const double MAX_BASE_TURRET_DEV = 1.0;
112
113    // Advance
114
115    const int ADV_SLOW_DIST = 60;
116    const int ADV_STOP_DIST = 6;
117    const int ADV_PER_SLOW_DIST = 12;
118    const int ADV_PER_STOP_DIST = 4;
119    const double ADV_SPEED = 75.0;
120    const double ADV_PER_SPEED = 20.0;
121    const double ADV_SPEED_SIGMA = 10.0;
122    const double ADV_SPEED_WT = 5.0;
123
124    // Advance slow
125
126    const int ADV_SLOW_STOP_DIST = 5;
127    const double ADV_SLOW_SPEED = 20.0;
128    const double ADV_SLOW_SPEED_SIGMA = 5.0;
129    const double ADV_SLOW_SPEED_WT = 5.0;
130
131    // Corridor advance
132
133    const int CORRIDOR_SPEED = 25;
134    const int CORRIDOR_SPEED_WIDE = 75;
135
136    // Maintain heading
137
138    const double MH_TURN_SIGMA = 45.0;
139    const double MH_TURN_WT = 1.0;
140
141    // Maintain transit heading
142
143    const double MTH_TURN_SIGMA = 45.0;
144    const double MTH_TURN_WT = 1.0;
145
146    // Avoid
147
148    const int AVOID_DIST = 36;
149    const double AVOID_TURN_SIGMA = 22.5;
150    const double AVOID_WT_FACTOR = 6.0;
151
152    // Transit avoid
153
154    const double TRANSIT_AVOID_TURN_SIGMA = 10.0;
155
156    // Avoid bias
157
158    const int AVOID_BIAS_DIST = 10;
159    const double AVOID_BIAS_ANGLE = 45.0;
160    const double AVOID_BIAS_SIGMA = 45.0;
161    const double AVOID_BIAS_WT = 1.0;
162
163    // Follow wall
164
165    const int FOLLOW_ABORT = 20;
166    const int FOLLOW_MAX_ALIGN_DIST = 40;
167    const int FOLLOW_STOP_DIST = 20;
168    const double FOLLOW_TURN_FACTOR = 0.2;
```

```
169    const double FOLLOW_TURN_SIGMA = 5.0;
170    const double FOLLOW_WT = 2.0;
171
172    // Maintain distance
173
174    const int DESIRED_DIST = 10;
175    const double MD_TURN_FACTOR = 0.2;
176    const double MD_TURN_SIGMA = 3.0;
177    const double MD_WT = 4.0;
178
179    // Follow path
180
181    const double NAV_MIN_ACT = 0.5;
182    const double NAV_SIGMA = 45.0;
183    const double NAV_WT = 5.0;
184
185    // Goal orient
186
187    const double GOAL_SIGMA = 45.0;
188    const double GOAL_WT = 5.0;
189
190    // Goal corridor orient
191
192    const double GOAL_CORRIDOR_NOISE = 5.0;          // Noise in turn angle
193
194    // Center home
195
196    const int CENTER_SPEED = 10;
197    const double CENTER_ERR_THRESH = 0.01;
198
199    // Angle localization
200
201    const int ANGLE_LOC_STEP = 10;
202    const int ANGLE_LOC_NUM_STEPS = 10;
203    const int ANGLE_LOC_SLEEP = 1;
204
205    /********** SEQUENCER CONSTANTS **********/
206
207    // 22.5 degrees between sonars
208
209    const int SONAR_SWEEP_WIDTH = 22;
210
211    // Sonar sweep speed
212
213    const int SONAR_SWEEP_SPEED = 20;
214
215    // Sonar sweep step (degrees)
216
217    const int SONAR_SWEEP_STEP = 2;
218
219    // Sonar sweep pause between steps (microseconds)
220
221    const unsigned int SONAR_SWEEP_PAUSE = 100000;
222
223    // Laser sweep step (degrees)
224
225    const int LASER_SWEEP_STEP = 5;
226
```

```
227    // Laser-limited sonar sweep speed
228
229    const int LLS_TURN_RATE = 150;
230
231    // Identification confirmation sequence (inches)
232
233    const double MAX_CONFIRM_DIST = 1.0;
234
235    // Local navigation sequencer tolerance (inches)
236
237    const double LOCAL_NAV_TOLERANCE = 18.0;
238
239    // Local navigation maximum timesteps for stall
240
241    const int STALL_TIMEOUT = 20;
242
243    // Angle above which local navigation turns robot in place
244
245    const double LOCAL_TIP_ANGLE = 90.0;
246
247    /********** CONTROL INTERFACE PARAMETERS **********/
248
249    const int NUM_CMD = 2;                   // Number of command outputs
250
251    enum { SPEED, TURN };                    // Command indexes
252
253    // Behavior modes
254
255    enum { EXPLORE_MODE, EXPLORE_LLS_MODE, NAVIGATION_MODE, TEST_MODE };
256
257    // Control commands
258
259    enum { CMD_EXPLORE,
260          CMD_NAV, CMD_NAV_KBD, CMD_STOP, CMD_SAVE, CMD_LOAD,
261          CMD_REDRAW, CMD_BUILD_GRID, CMD_SAVE_GRID, CMD_LOAD_GRID,
262          CMD_GRID_IDENT, CMD_GRID, CMD_CLEAR, CMD_CLEAR_ROBOT,
263          CMD_SONAR_SCAN, CMD_SONAR_SWEEP, CMD_SONAR_SWEEP_ABS,
264    CMD_CLEAR_SONAR,
265          CMD_LASER_SCAN, CMD_LASER_SWEEP, CMD_LASER_SWEEP_ABS,
266    CMD_CLEAR_LASER,
267          CMD_LLS_SCAN, CMD_LLS_SWEEP, CMD_LLS_SWEEP_ABS, CMD_CLEAR_LLS,
268          CMD_GRID_UNDO, CMD_CENTER, CMD_PLACE_MAP,
269          CMD_PLACE_IDENT, CMD_PLACE_GRID,
270          CMD_LOCAL_NAV, CMD_ADD_PLACE, CMD_EDIT_PLACE,
271          CMD_ADD_EDIT_LINK, CMD_DELETE_LINK,
272          CMD_CLEAR_GLOBAL, CMD_SAVE_GLOBAL, CMD_LOAD_GLOBAL,
273          CMD_DISPLAY_GLOBAL, CMD_GLOBAL_UNDO, CMD_INTEGRATE_GRID,
274          CMD_FIND_FRONTIERS, CMD_DISPLAY_EDGES, CMD_DISPLAY_FRONTIERS,
275          CMD_GOTO_FRONTIER, CMD_UPDATE_NAV_GRID, CMD_DETECT_CORRIDORS,
276          CMD_CONNECT_CL, CMD_SEND_CL_GRID,
277          CMD_BUMP,
278          CMD_INIT_COMM, CMD_SEND_MSG, CMD_RECEIVE_MSG,
279          CMD_EXIT };
280
281    /********** GRAPHICS CONSTANTS **********/
282
283    // Control window graphic parameters
284
```

```cpp
285     const int CON_NUM_CMD = CMD_EXIT + 1;      // Number of command buttons
286
287     const int CON_WIN_LEFT = 850;          // x-coord of left side
288     const int CON_WIN_TOP = 0;             // x-coord of top
289
290     // Number of button columns
291     const int CON_COLS = 2;
292
293     // Number of button rows
294     const int CON_ROWS = (int) ((double) CON_NUM_CMD / 2.0 + 0.5);
295
296     const int CON_BUTTON_WIDTH = 150;   // Button width
297     const int CON_BUTTON_HEIGHT = 25;   // Button height
298
299     const int CON_LAB_WIDTH = 130;              // Label width
300                                  // (Must be less than button width)
301     const int CON_LAB_HEIGHT = 10;              // Label height
302                                  // (Must be less than button height)
303
304     // Evidence grid window screen boundaries
305
306     const int EGWIN_LEFT = 420;
307     const int EGWIN_TOP = 0;
308     const int EGWIN_RIGHT = 932;
309     const int EGWIN_BOTTOM = 512;
310
311     // Evidence grid window world coordinate boundaries
312
313     const int EGWIN_WC_LEFT = -3000;
314     const int EGWIN_WC_BOTTOM = -3000;
315     const int EGWIN_WC_RIGHT = 3000;
316     const int EGWIN_WC_TOP = 3000;
317
318     // Navigation grid window screen boundaries
319
320     const int NAV_WIN_LEFT = 420;
321     const int NAV_WIN_TOP = 0;
322     const int NAV_WIN_RIGHT = 932;
323     const int NAV_WIN_BOTTOM = 512;
324
325     // Navigation grid window world coordinate boundaries
326
327     const int NAV_WIN_WC_LEFT = -6000;
328     const int NAV_WIN_WC_BOTTOM = -6000;
329     const int NAV_WIN_WC_RIGHT = 6000;
330     const int NAV_WIN_WC_TOP = 6000;
331
332     /*
333     const int NAV_WIN_WC_LEFT = -3000;
334     const int NAV_WIN_WC_BOTTOM = -3000;
335     const int NAV_WIN_WC_RIGHT = 3000;
336     const int NAV_WIN_WC_TOP = 3000;
337     */
338
339     // Global evidence grid window screen boundaries
340
341     const int GLOBAL_WIN_LEFT = 0;
342     const int GLOBAL_WIN_TOP = 0;
```

```
343    const int GLOBAL_WIN_RIGHT = 1024;
344    const int GLOBAL_WIN_BOTTOM = 1024;
345
346    /*
347    const int GLOBAL_WIN_LEFT = 0;
348    const int GLOBAL_WIN_TOP = 0;
349    const int GLOBAL_WIN_RIGHT = 512;
350    const int GLOBAL_WIN_BOTTOM = 512;
351    */
352
353    // Evidence grid window world coordinate boundaries
354
355    const int GLOBAL_WIN_WC_LEFT = -6000;
356    const int GLOBAL_WIN_WC_BOTTOM = -6000;
357    const int GLOBAL_WIN_WC_RIGHT = 6000;
358    const int GLOBAL_WIN_WC_TOP = 6000;
359
360    /*
361    const int GLOBAL_WIN_WC_LEFT = -3000;
362    const int GLOBAL_WIN_WC_BOTTOM = -3000;
363    const int GLOBAL_WIN_WC_RIGHT = 3000;
364    const int GLOBAL_WIN_WC_TOP = 3000;
365    */
366
367    // Color of robot in global window
368
369    static char GLOBAL_ROBOT_COLOR[ STRLEN] = "Blue";
370
371    // Color of contact marker in global window
372
373    static char CONTACT_COLOR[ STRLEN] = "Red";
374
375    // Size of contact marker in global window
376
377    const int CONTACT_MARK_SIZE = 50;
378
379    /********** FRONTIER CONSTANTS *********/
380
381    // Maximum number of frontiers
382
383    const int MAX_FRONTIERS = 1000;
384
385    // Radius of neighborhood around visited frontier (1/10 inch)
386
387    const double VISIT_RADIUS = 60.0;
388
389    // Radius of neighborhood around inaccesible frontier (1/10 inch)
390
391    const double INAC_RADIUS = 120.0;
392
393    // Maximum number of colors for blob coloring
394
395    const int MAX_COLORS = 1000;
396
397    // Number of colors to display
398
399    const int DISPLAY_COLORS = 16;
400
```

```cpp
401     // Radius of region centroid marker
402
403     const double CENTROID_MARK_RADIUS = 75.0;
404
405     // Minimum region size
406
407     const int MIN_REGION_SIZE = 6;
408     // const int MIN_REGION_SIZE = 1;
409
410     // Maximum frontier distance
411
412     const double MAX_DIST = 100000.0;
413
414     // Frontier edge color
415
416     static char EDGE_COLOR[ STRLEN]  = "red";
417
418     // Color table
419
420     static char color_table[ DISPLAY_COLORS][ STRLEN]  = {
421       "Blue", "Green", "Gold", "Red",
422       "SkyBlue", "LimeGreen", "Orange", "Magenta",
423       "RoyalBlue", "Cyan", "LightCoral", "Violet",
424       "SteelBlue", "Aquamarine", "Purple", "Turquoise" } ;
425
426     // Color conversions for robot window
427
428     static int robot_color[ DISPLAY_COLORS]  = {
429       1, 6, 11, 16,
430       2, 7, 12, 17,
431       3, 8, 13, 18,
432       4, 9, 15, 20 } ;
433
434     /********** NAVIGATION CONSTANTS **********/
435
436     // Distance to retreat on bumper contact (1/10 inch)
437
438     const int BUMP_RECOIL = 20;
439
440     // Speed of bump recoil
441
442     const int BUMP_RECOIL_SPEED = 20;
443
444     // Search status codes
445
446     const int SEARCH_SUCCESS = 0;
447     const int SEARCH_FAIL = 1;
448     const int SEARCH_TIMEOUT = 2;
449
450     // Maximum number of cells to search
451
452     const int SEARCH_MAX_CELLS = 10000;
453
454     // Maximum number of obstacle cells allowed in path region
455
456     const int MAX_OBS_COUNT = 2;
457
458     // Color of path in grid/global window
```

```
459
460    static char PATH_COLOR[ STRLEN]  = "Red";
461    static char OPT_PATH_COLOR[ STRLEN]  = "Blue";
462    static char TRAV_PATH_COLOR[ STRLEN]  = "Red";
463
464    // Color of path in robot window
465
466    const int ROBOT_PATH_COLOR = 16;     // Red
467    const int ROBOT_OPT_PATH_COLOR = 2; // Blue
468    const int ROBOT_TRAV_PATH_COLOR = 16;     // Red
469
470    // Waypoint lookhead window (# waypoints)
471
472    const int WAYPOINT_WINDOW = 5;
473
474    // Number of waypoints between LLS sweeps during navigation
475
476    const int NAV_LLS_SWEEP_INTERVAL = 10000; // Never
477
478    /********** CORRIDOR CONSTANTS **********/
479
480    // Number of sensors to either side of sensor to check
481
482    const int CORRIDOR_SPAN = 3;
483
484    // Amount of forward space needed to be clear
485
486    const int CORRIDOR_FWD_RANGE = 12;
487
488    // Amount of space needed on sides of robot
489
490    const int CORRIDOR_SIDE_CLEARANCE = 6;
491
492    // Amount of forward space for wide corridor
493
494    const int CORRIDOR_WIDE_FWD_RANGE = 48;
495
496    // Amount of side space for a wide corridor
497
498    const int CORRIDOR_WIDE_SIDE_CLEARANCE = 24;
499
500    // Maximum deviation between corridor angle and goal bearing
501
502    const double CORRIDOR_MAX_DEVIATION = 90.0;
503
504    // Color of corridor in global window
505
506    static char CORRIDOR_COLOR[ STRLEN]  = "Blue";
507    static char CORRIDOR_WIDE_COLOR[ STRLEN]  = "Green";
508    static char CORRIDOR_SELECT_COLOR[ STRLEN]  = "Red";
509    static char CORRIDOR_SELECT_WIDE_COLOR[ STRLEN]  = "Gold";
510
511    // Color of corridor in robot window
512
513    const int CORRIDOR_COLOR_ROBOT = 2;                    // Blue
514    const int CORRIDOR_WIDE_COLOR_ROBOT = 6;          // Green
515    const int CORRIDOR_SELECT_COLOR_ROBOT = 14;          // DeepPink
516    const int CORRIDOR_SELECT_WIDE_COLOR_ROBOT = 11;       // Yellow
```

```
517
518    /********** CONTINUOUS LOCALIZATION DECLARATIONS **********/
519
520    // Continuous localization host
521
522    static char CONTLOC_HOST[ STRLEN]  = "sun28";
523
524    // Continuous localization communication channel ID
525
526    const int CONTLOC_CHANNEL = 0;
527
528    // Minimum number of occupied cells for usable map
529
530    const int CONTLOC_MIN_OCC = 0;
531
532    // Exploration relocalization interval (inches)
533
534    const int EXPLORE_RELOC_DISTANCE = 96;
535
536    // Navigation relocalization interval (inches)
537
538    const int NAV_RELOC_DISTANCE = 24;
539
540    /********** MULTIROBOT DECLARATIONS **********/
541
542    // Message indicating new map exists
543
544    static char NEW_MAP_MSG[ STRLEN]  = "NEWMAP";
545
546    /********** MISCELLANEOUS DECLARATIONS **********/
547
548    // Status codes
549
550    const int OK = 0;
551    const int ALT = 1;
552    const int RETRY = 2;
553    const int ABORT = -1000;
554    const int TIMEOUT = -1001;
555    const int NO_PATH = -1002;
556    const int NO_FRONTIERS = -1003;
557
558    // External C functions
559
560    extern "C"  int abs(int);
561    extern "C"  int sleep(int);
562    extern "C"  int usleep(unsigned int);
563    extern "C"  void exit(int);
564    extern "C"  void registergrids(Map3D map1, Map3D map2, double *dx,
565                              double *dy, double *dt, double *fitness);
566
567    // Number of moving obstacles
568
569    const int NUM_MOB = 0;
570
571    // Length of experimental trial (steps)
572
573    //const int TRIAL_LENGTH = 10000;
574
```

```
575     const int TRIAL_LENGTH = 50000;
576
577     //const int TRIAL_LENGTH = 1000000000;    // Run forever
578
579     // Margin for random robot placement
580
581     const int RAND_MARGIN = 200;
582
583     class agent {
584
585     public:
586        agent(void);                          // Constructor
587        void control(void);                   // Main control loop
588
589     private:
590        int bumped[ NUM_TOUCH] ;              // BUMPER HACK ARRAY
591
592        robot r;                      // Controlled robot
593        arbitrator *speed_arb;                // Speed command arbitrator
594        arbitrator *turn_arb;                 // Turn command arbitrator
595        place_net pnet;               // Place network
596        char apndir[ STRLEN] ;                // Name of APN directory
597
598        mobstacle mob_list[ NUM_MOB + 1] ;    // Moving obstacles
599
600        Map3D global_grid;                    // Global evidence grid
601        Map3D old_global;                     // Old global evidence grid
602        Map3D egrid;                          // Evidence grid
603        Map3D old_grid;               // Old evidence grid
604        Map3D edge_grid;                      // Frontier edge grid
605        Map3D nav_grid;               // Navigation grid
606
607        int region_map[ GLOBAL_X_RES] [ GLOBAL_Y_RES] ;    // Colored region grid
608
609        int visit[ NAV_X_RES] [ NAV_Y_RES] ;  // Visit map for path planning
610
611        frontier frontiers[ MAX_FRONTIERS] ;     // List of frontiers
612        int num_front;                // Number of frontiers
613
614        frontier front_visit[ MAX_FRONTIERS] ;   // Visited frontiers
615        int num_visit;                // Number of visited frontiers
616
617        frontier front_inac[ MAX_FRONTIERS] ;    // Inaccessible frontiers
618        int num_inac;                         // Number of inaccessible frontiers
619
620        int corridor[ NUM_RANGE] ;            // Corridor detection array
621        int wide_corridor[ NUM_RANGE] ;          // Wide corridor detection
622     array
623
624        CylSensorModelArray sonar_smd;     // Sonar sensor model
625        CylSensorModelArray sonar_clear_smd;    // Sonar sensor model (clear)
626
627        control_panel control_window;            // Control window
628     //  bar_graph speed_window;          // Speed command display
629     //  bar_graph turn_window;           // Turn command display
630        window *grid_window;                 // Evidence grid window (pointer)
631     //  window *nav_window;                  // Navigation grid window (pointer)
632        window *global_window;               // Global grid window (pointer)
```

```
633
634    int global_refresh;                    // Set when global grid is displayed
635    int realtime_display;                  // Flags whether to display robot
636
637    ofstream *logfile;                     // Log file
638
639    int multi_mode;               // Multirobot mode (0:single, 1:multi)
640    int contloc_mode;                      // Continuous localization mode
641    int behavior_mode;                     // Behavior mode
642    int home_dist;                // Path distance from home
643
644    int destin;                            // Destination index
645
646    int timer;                             // Total elapsed time (steps)
647
648    double cpu_volt;                       // CPU battery voltage
649    double motor_volt;                     // Motor battery voltage
650    double cpu_min;               // Minimum CPU battery voltage
651    double motor_min;                      // Minimum motor battery voltage
652
653    double transit_dir;                    // Transit direction
654
655    int pause_mode;
656
657    int cell_count;               // Number of cells searched
658
659    void reset(void);                      // Reset position and timer
660    int user_command(void);                // Execute user command (if any)
661    int iscan(void);                       // Scan for interrupt
662    void terminate(void);                  // End session
663    void power_check(void);                // Check battery power
664
665    // Behavior modes
666
667    void bump_test(void);                  // Bumper test
668    void manual_exploration(void);    // Map territory under manual
669  control
670    void exploration(void);                // Explore uncharted territory
671    void exploration_lls(void);            // Explore uncharted territory (LLS)
672    void reactive_exploration(void);  // Explore uncharted territory
673    // reactively
674    void navigation(void);                 // Navigate to destination (mouse)
675    void navigation_keyboard(void);   // Navigate to destination
676  (keyboard)
677    void local_navigation(void);              // Navigate to local
678  destination point
679    void frontier_navigate(void);             // Navigate to frontier
680  centroid
681
682    // Explore uncharted territory (multiple trials)
683    void multi_exploration(void);
684
685    // Explore uncharted territory reactively (multiple trials)
686    void multi_reactive_exploration(void);
687
688    // Behavior sequencers
689
690    // Manual exploration sequencer
```

```
691         void manual_exploration_seq(void);
692
693         // Exploration sequencer
694         void exploration_seq(void);
695
696         // Exploration sequencer (laser-limited sonar)
697         void exploration_lls_seq(void);
698
699         // Reactive exploration sequencer
700         void reactive_exploration_seq(void);
701
702         int navigation_seq(void);          // Navigation sequencer
703         void search_seq(void);             // Search sequencer
704         void map_seq(void);                // Build local grid
705         void center_seq(void);             // Move to center of current place
706
707         // Local navigation sequencer
708         int local_nav_seq(int x, int y);   // Local destination coordinates
709
710         // Local navigation sequencer for path following
711         int path_local_nav_seq(path p,             // Path to folloq
712                         int &waypoint);    // Index of next waypoint
713
714         // Local navigation sequencer (continuous motion)
715         int local_cont_nav_seq(int x, int y);    // Local destination coords
716
717         // Local navigation sequencer (with alternate goal)
718         int local_nav_seq_alt(int gx, int gy,         // Goal coordinates
719                         int ax, int ay);   // Alternate goal coordinates
720
721         // Navigate to goal by planning and following path
722         int path_nav_seq(double gx, double gy); // World coords of goal
723
724         // Navigate to frontier by planning and following path
725         int frontier_path_nav_seq(int front_index);    // Frontier index
726
727         // Place identification sequencer
728         void ident_seq(void);
729
730         // Grid identification sequencer
731         void grid_ident_seq(void);
732
733         // Rotate and reset DR angle to match stored range image
734         void angle_loc_seq(int image[ NUM_RANGE] );
735
736         // Translate to match stored range image
737         void trans_loc_seq(int image[ NUM_RANGE] );
738
739         // Rotate sonar sensors and scan
740         void sonar_sweep_seq(Map3D map);
741
742         // Rotate sonar sensors and scan (absolute coordinates)
743         void sonar_sweep_abs_seq(Map3D map);
744
745         // Rotate laser scanner and scan
746         void laser_sweep_seq(Map3D map);
747
748         // Rotate laser scanner and scan (absolute coordinates)
```

```
749    void laser_sweep_abs_seq(Map3D map);
750
751    // Laser-limited sonar sweep
752    void lls_sweep_seq(Map3D map);
753
754    // Laser-limited sonar sweep (absolute coordinates)
755    void lls_sweep_abs_seq(Map3D map);
756
757    // Navigate to selected frontier
758    int frontier_nav_seq(int front_index);   // Frontier destination index
759
760    // Behavior sets
761
762    // Behavior set for reactive exploration
763    int reactive_explore_behaviors(void);
764
765    int navigation_behaviors(void);          // Behavior set for navigation
766
767    // Behavior set for local navigation
768    int local_navigation_behaviors(int gx, int gy);
769
770    // Behaviors
771
772    /********** LOW-LEVEL BEHAVIORS **********/
773
774    void bump_halt(void);        // Go limp if bumper touched
775    void recoil(void);           // If touched in forward half, move
776 backward
777    void bump_recoil(void);             // If bumper contact, recoil away
778    void wander(void);           // Make small random turns
779    int advance(void);           // Move forward unless front is blocked
780    int advance_slow(void);      // Move forward slowly unless front is
781 blocked
782
783    // Realign turret if it is not aligned with base
784    void maintain_alignment(void);
785
786    // Avoid nearby obstacles
787    void avoid(void);
788
789    // If front is completely blocked, bias avoidance toward the left side
790    void avoid_bias_left(void);
791
792    // If front is completely blocked, bias avoidance toward the right
793 side
794    void avoid_bias_right(void);
795
796    // Maintain current heading
797    void maintain_heading(void);
798
799    void veer(void);        // Veer away from potential collisions
800
801    void follow_wall_right(void);    // Align with right wall
802    void follow_wall_left(void);     // Align with left wall
803
804    // Maintain desired distance from right wall
805    void maintain_distance_right(void);
806
```

```
807     // Maintain desired distance from left wall
808     void maintain_distance_left(void);
809
810     // Turn toward goal
811     void goal_orient(int gx, int gy);
812
813     /********** NAVIGATION BEHAVIORS **********/
814
815     int follow_path(void);              // Turn to follow path
816     int detect_dest(int destin);               // Detect arrival at
817 destination
818
819     // Low-level commands
820
821     void set_defaults(void);     // Set default command values
822     void update(void);           // Update robot state and evidence grid
823     void execute(void);          // Send commands to robot
824
825     // Move obstacles
826     void move_obstacles(void);
827
828     // Delete obstacles
829     void del_obstacles(void);
830
831     // File access commands
832
833     void save_net(void);               // Save network in file
834     void load_net(void);               // Load network from file
835
836     /********** LOCALIZATION FUNCTIONS **********/
837
838     // Compute difference between image and range input
839     double compute_range_err(int image[NUM_RANGE], vector rinput);
840
841     // Compute translation vector between expected and actual position
842     void trans_loc_vector(int image[NUM_RANGE], int &dx, int &dy);
843
844     /********** EVIDENCE GRID FUNCTIONS **********/
845
846     // Display evidence grid in X window
847     void grid_display(window *win,      // Window pointer
848                 Map3D map);             // Evidence grid
849
850     // Display global evidence grid in X window
851     void grid_display_global(Map3D map);     // Evidence grid
852
853     // Display local grid for place
854     void display_place_grid(void);
855
856     // Display edge segments detected in evidence grid
857     void grid_display_edges(int grid[GLOBAL_X_RES][GLOBAL_Y_RES]);
858
859     // Display regions detected in evidence grid
860     void grid_display_regions(int grid[GLOBAL_X_RES][GLOBAL_Y_RES]);
861
862     // Display robot in window
863     void display_robot(window *win,     // Window
864                 int x, int y,           // Robot position (1/10 inch)
```

```
865                    int theta,         // Robot heading (1/10 degree)
866                    int turret); // Robot turret angle (1/10 degree)
867
868     /********** FRONTIER FUNCTIONS **********/
869
870     // Copy frontier <f2> to frontier <f1>
871     void frontier_copy(frontier &f1, frontier f2);
872
873     // Find frontiers in global grid
874     void find_frontiers(void);
875
876     // Find frontier edges in <raw> grid and store them in <edge> grid
877     void find_frontier_edges(Map3D *raw,          // Raw evidence grid
878  (pointer)
879                          Map3D *edge,        // Frontier edge grid
880  (pointer)
881                          double height);     // Z-axis of edge plane
882
883     // Find frontier regions in <edge> grid and add new frontiers
884     void find_frontier_regions(Map3D edge,   // Frontier edge grid
885                          double height);   // Z-coord of edge plane
886
887
888     // Segment <grid> image into regions in <color> using spreading
889  activation
890     void spread_segment(Map3D grid,          // Uncolored grid
891                     int color[ GLOBAL_X_RES][ GLOBAL_Y_RES] , // Colored grid
892                     double height);    // Z-coord of edge plane
893
894     // Print colored grid cell values
895     void print_region_map(int grid[ GLOBAL_X_RES][ GLOBAL_Y_RES] );     //
896  Colored grid
897
898     // Determine size and centroid of frontier regions
899     void analyze_regions(int grid[ GLOBAL_X_RES][ GLOBAL_Y_RES] );      //
900  Colored grid
901
902     // Check whether centroid corresponds to previously visited frontier
903     // Return 1 if visited, 0 otherwise
904     int agent::visited(double cx, double cy);
905                         // Centroid of new region
906
907     // Check whether centroid corresponds to inaccessible frontier
908     // Return 1 if inaccessible, 0 otherwise
909     int agent::inaccessible(double cx, double cy);
910                         // Centroid of new region
911
912     // Return index of unvisited, accessible frontier closest to (x, y)
913     // Return -1 if no such frontier exists
914     int closest_frontier(double x, double y);
915
916     // Mark region centroids in evidence grid window
917     void display_region_centroids(double cx,        // Display center x-
918  coord
919                          double cy); // Display center y-coord
920
921     // Mark region centroids in robot window
922     void display_robot_region_centroids(void);
```

```
923
924     // Check whether cell (x, y) is part of frontier <front_index>
925     int check_frontier_cell(int x, int y,    // Cell index
926                             int front_index);    // Frontier index
927
928
929     /****************** NAVIGATION FUNCTIONS ******************/
930
931     // Move forward if front corridor is clear
932     void corridor_advance(void);
933
934     // Turn toward clear corridor closest to goal bearing
935     void goal_corridor_orient(int gx, int gy);
936
937     // Update navigation grid based on global grid
938     void update_nav_grid(void);
939
940     // Plan path to goal location (return 1 if path found, 0 otherwise)
941     int path_plan(double wx, double wy,              // World coords of goal
942               path &nav_path);          // Navigation path (optimized)
943
944     // Plan path to goal location (return 1 if path found, 0 otherwise)
945     int frontier_path_plan(double wx, double wy, // World coords of goal
946                       int front_index,       // Frontier index
947                       path &nav_path);       // Navigation path
948
949     // Print all cells on path
950     void print_path(path p);
951
952     // Draw path in window
953     void display_path(path p,           // Path
954                   char *pcolor, // Path color
955                   window *win); // Window
956
957     // Draw path in robot window
958     void display_path_robot(path p,          // Path
959                       int pcolor);          // Path color
960
961     // Find path from (sx, sy) to (gx, gy)
962     int find_path(int sx, int sy,              // Start cell
963             int gx, int gy,         // Goal cell
964             path &p);           // Path
965
966     // Find path from (sx, sy) to (gx, gy) or any point on frontier
967 <front_index>
968     int frontier_find_path(int sx, int sy,  // Start cell
969                     int gx, int gy,  // Goal cell
970                     int front_index, // Frontier index
971                     path &p);        // Path
972
973     // Search cell (x,y) and return search status
974     int search_cell(int x, int y,             // Search cell
975                 int gx, int gy, // Goal cell
976                 path &p);       // Path
977
978     // Search cell (x,y) while navigating to frontier and return search
979 status
980     int frontier_search_cell(int x, int y,          // Search cell
```

```
981                              int gx, int gy,              // Goal cell
982                              int front_index,             // Frontier index
983                              path &p);                    // Path
984
985        // Find index of (unvisited) neighbor closest to goal
986        int closest_neighbor(int x, int y,              // Current cell index
987                             int gx, int gy,       // Goal cell index
988                             int &nx, int &ny);     // Next cell index
989
990        // Reverse order of steps on path
991        void reverse_path(path old_path,         // Initial path
992                          path &new_path);       // Reversed path
993
994
995        // Optimize path by jumping between adjacent path cells
996        void optimize_path(path old_path, // Initial path
997                           path &new_path);    // Optimized path
998
999        // Convert path in grid cell coords to world coords
1000       void generate_world_path(path grid_path,        // Path in nav grid
1001                                path &world_path);   // Path in world coords
1002
1003       // Initialize path
1004       void path_init(path &p);      // Path
1005
1006       // Add point to path
1007       void path_add(path &p,         // Path
1008                     int x, int y);    // Point to add to path
1009
1010       //   Check to see whether region around point is free of known
1011   obstacles
1012       int check_clear(int x, int y);
1013
1014       //   Check to see whether region around point overlaps frontier
1015       int check_frontier_arrival(int x, int y, int front_index);
1016
1017       // Finds waypoint furthest on path within destination tolerance, or
1018       // waypoint on path <p> closest to (x, y), returning the distance
1019       // to that point, and the waypoint's index in <index>
1020       double closest_waypoint(path p,           // Path
1021                               int x, int y,     // Current position (1/10
1022   inch)
1023                               int index,        // Index of current waypoint
1024                               int &close_index);   // Index of closest waypoint
1025
1026       /****************** CORRIDOR FUNCTIONS ******************/
1027
1028       // Detect freespace cooridors in vicinity of robot
1029       void detect_corridors(void);
1030
1031       // Check whether a corridor exists centered around sensor <center>
1032       // Return 1 if true, 0 otherwise
1033       int check_corridor(int center,     // Index of sensor in center of
1034   corridor
1035                          int fwd_range,    // Required forward space
1036                          int side_clear);  // Required side space
1037
1038
```

```
1039        // Check whether <sensor> is clear for cooridor <center>
1040        int corridor_check_sensor(int center,          // Center sensor index
1041                               int sensor,          // Sensor index
1042                               int fwd_range,       // Required fwd space
1043                               int side_clear);     // Required side space
1044
1045        // Display corridors in robot window
1046        void display_corridors(void);
1047
1048        // Display corridor boundaries centered around sensor <center>
1049        void display_corridor(window *win,         // Window
1050                           int center, // Center sensor index
1051                           int fwd_range,    // Required forward space
1052                           int side_clear,   // Required side space
1053                           char *color);     // Corridor color
1054
1055        // Display corridor boundaries centered around sensor <center> in
1056    robot window
1057        void display_corridor_robot(int center, // Center sensor index
1058                               int fwd_range,  // Required forward space
1059                               int side_clear, // Required side space
1060                               int color); // Corridor color
1061
1062        // Select corridor nearest to specified heading
1063        int select_corridor(double heading);    // Heading (degrees)
1064
1065        /********** INTERFACE TO CONTINUOUS LOCALIZATION **********/
1066
1067        // Initialize communications with continuous localization
1068        void connect_cl(void);
1069
1070        // Send global grid to continuous localization
1071        void send_cl_grid(void);
1072
1073        /********** MULTIROBOT COMMUNICATION **********/
1074
1075        // Initialize robot communication channel
1076        void init_robot_comm(void);
1077
1078        // Send message to other robot
1079        void send_robot_message(char *message);
1080
1081        // Send message to other robot (user mode)
1082        void user_send_robot_message(void);
1083
1084        // BEGIN SCOUT THESIS CHANGE
1085
1086        // Receive message from other robot
1087        // Returns 1 if message received, 0 otherwise
1088        int receive_robot_message(int channel, char *message);
1089
1090        // END SCOUT THESIS CHANGE
1091
1092        // Receive message from other robot (user mode)
1093        void user_receive_robot_message(void);
1094
1095        /********** MULTIROBOT EXPLORATION **********/
1096
```

```
1097        // Integrate new map from remote robot (if a new map exists)
1098        void integrate_remote_map(void);
1099     };
1100
1101     #endif
```

APPENDIX H. FRONTIER-BASED EXPLORATION CODE – AGENT.CC

This appendix contains the source code for the routine that controls the robot's exploration behaviors.

```
1    /*
2
3       agent.cc
4
5       Agent class
6       Original code by Brian Yamauchi
7
8       Modifications for SCOUT THESIS
9       By Patrick A. Hillmeyer
10
11   */
12
13   #include <iostream.h>
14   #include <math.h>
15   #include <string.h>
16
17   #include "agent.h"
18
19   // Arc direction strings
20
21   const char dir_str[ 16][ 20]  =
22   { "forward", "fwd-fwd-lf", "fwd-lf", "fwd-lf-lf",
23       "left", "back-lf-lf", "back-lf", "back-back-lf",
24       "back", "back-back-rt", "back-rt", "back-rt-rt",
25       "right", "fwd-rt-rt", "fwd-rt", "fwd-fwd-rt" } ;
26
27   const char voice_str[ 16][ STRLEN]  =
28   { "forward 0.", "forward 1.", "forward left 2.", "left 3.",
29       "left 4.", "left 5.", "back left 6.", "back 7.",
30       "back 8 .", "back 9.", "back right 10.", "right 11.",
31       "right 12.", "right 13.", "forward right 14.", "forward 15." } ;
32
33   /********** AGENT CLASS CONSTRUCTOR **********/
34
35   agent::agent(void)
36   {
37
38       char Global_Grid[ STRLEN] ;        // Global Grid label
39       char Control_Panel[ STRLEN] ;            // Control Panel Label
40
41       // Constructor
42
43       char labels[ MAX_CON][ CON_LEN] ;
44       int i;
45
46       // Initialize mode flags
47
48       multi_mode = 0;
49       behavior_mode = EXPLORE_MODE;
50       contloc_mode = 0;
51       home_dist = 0;
52       destin = 0;
```

```
53
54        // Initialize graphics flags
55
56        global_refresh = 1;
57        realtime_display = 1;
58
59        // Initialize frontier counters
60
61        num_front = 0;
62        num_inac = 0;
63
64        // Initialize power variables
65
66        cpu_volt = CPU_FULL_VOLTAGE;
67        motor_volt = MOTOR_FULL_VOLTAGE;
68
69        cpu_min = cpu_volt;
70        motor_min = motor_volt;
71
72        // Initialize abitrator windows
73
74        speed_arb = new arbitrator(SPEED_RES, SPEED_MIN, SPEED_MAX,
75    SPEED_DEF, 0,
76                                  SPEED_NOISE);
77        if (speed_arb == NULL) {
78          cout << "agent::agent: Unable to allocate space for speed
79    arbitrator."
80                << endl;
81          exit(-1);
82        }
83
84        turn_arb = new arbitrator(TURN_RES, TURN_MIN, TURN_MAX, TURN_DEF, 1,
85                               TURN_NOISE);
86        if (turn_arb == NULL) {
87          cout << "agent::agent: Unable to allocate space for turn
88    arbitrator."
89                << endl;
90          exit(-1);
91        }
92
93        //      speed_window.init(SPWIN_X, SPWIN_Y, SPWIN_WIDTH, SPWIN_HEIGHT,
94    "Speed",
95        //                       SPEED_RES, SPWIN_MIN, SPWIN_MAX);
96        //      turn_window.init(TUWIN_X, TUWIN_Y, TUWIN_WIDTH, TUWIN_HEIGHT,
97    "Turn",
98        //                       TURN_RES, TUWIN_MIN, TUWIN_MAX);
99
100       // Initialize control window
101
102       strcpy(labels[ CMD_EXPLORE] , "EXPLORE" );
103       strcpy(labels[ CMD_NAV] , "NAVIGATE" );
104       strcpy(labels[ CMD_NAV_KBD] , "NAVIGATE (KBD)" );
105       strcpy(labels[ CMD_STOP] , "STOP" );
106       strcpy(labels[ CMD_SAVE] , "SAVE APN" );
107       strcpy(labels[ CMD_LOAD] , "LOAD APN" );
108       strcpy(labels[ CMD_REDRAW] , "DISPLAY APN" );
109       strcpy(labels[ CMD_BUILD_GRID] , "BUILD GRID" );
110       strcpy(labels[ CMD_SAVE_GRID] , "SAVE GRID" );
```

```
111     strcpy(labels[ CMD_LOAD_GRID] , "LOAD GRID");
112     strcpy(labels[ CMD_GRID] , "DISPLAY GRID");
113     strcpy(labels[ CMD_CLEAR] , "CLEAR GRID");
114     strcpy(labels[ CMD_CLEAR_ROBOT] , "CLEAR ROBOT (ABS)");
115     strcpy(labels[ CMD_SONAR_SCAN] , "SONAR SCAN");
116     strcpy(labels[ CMD_SONAR_SWEEP] , "SONAR SWEEP");
117     strcpy(labels[ CMD_SONAR_SWEEP_ABS] , "SONAR SWEEP (ABS)");
118     strcpy(labels[ CMD_CLEAR_SONAR] , "CLEAR + SONAR SWEEP");
119     strcpy(labels[ CMD_LASER_SCAN] , "LASER SCAN");
120     strcpy(labels[ CMD_LASER_SWEEP] , "LASER SWEEP");
121     strcpy(labels[ CMD_LASER_SWEEP_ABS] , "LASER SWEEP (ABS)");
122     strcpy(labels[ CMD_CLEAR_LASER] , "CLEAR + LASER SWEEP");
123     strcpy(labels[ CMD_LLS_SCAN] , "LLS SCAN");
124     strcpy(labels[ CMD_LLS_SWEEP] , "LLS SWEEP");
125     strcpy(labels[ CMD_LLS_SWEEP_ABS] , "LLS SWEEP (ABS)");
126     strcpy(labels[ CMD_CLEAR_LLS] , "CLEAR + LLS SWEEP");
127     strcpy(labels[ CMD_GRID_UNDO] , "UNDO SCAN/SWEEP");
128     strcpy(labels[ CMD_GRID_IDENT] , "GRID IDENT");
129     strcpy(labels[ CMD_CENTER] , "PLACE CENTER");
130     strcpy(labels[ CMD_PLACE_MAP] , "PLACE MAP");
131     strcpy(labels[ CMD_PLACE_IDENT] , "PLACE IDENT");
132     strcpy(labels[ CMD_PLACE_GRID] , "DISPLAY PLACE GRID");
133     strcpy(labels[ CMD_LOCAL_NAV] , "LOCAL NAVIGATION");
134     strcpy(labels[ CMD_ADD_PLACE] , "ADD PLACE");
135     strcpy(labels[ CMD_EDIT_PLACE] , "EDIT PLACE");
136     strcpy(labels[ CMD_ADD_EDIT_LINK] , "ADD/EDIT LINK");
137     strcpy(labels[ CMD_DELETE_LINK] , "DELETE LINK");
138     strcpy(labels[ CMD_CLEAR_GLOBAL] , "CLEAR GLOBAL GRID");
139     strcpy(labels[ CMD_SAVE_GLOBAL] , "SAVE GLOBAL GRID");
140     strcpy(labels[ CMD_LOAD_GLOBAL] , "LOAD GLOBAL GRID");
141     strcpy(labels[ CMD_DISPLAY_GLOBAL] , "DISPLAY GLOBAL GRID");
142     strcpy(labels[ CMD_GLOBAL_UNDO] , "UNDO GLOBAL CHANGES");
143     strcpy(labels[ CMD_INTEGRATE_GRID] , "INTEGRATE LOCAL GRID");
144     strcpy(labels[ CMD_FIND_FRONTIERS] , "FIND FRONTIERS");
145     strcpy(labels[ CMD_DISPLAY_EDGES] , "DISPLAY EDGES");
146     strcpy(labels[ CMD_DISPLAY_FRONTIERS] , "DISPLAY FRONTIERS");
147     strcpy(labels[ CMD_GOTO_FRONTIER] , "GO TO FRONTIER");
148     strcpy(labels[ CMD_UPDATE_NAV_GRID] , "UPDATE NAV GRID");
149     strcpy(labels[ CMD_DETECT_CORRIDORS] , "DETECT CORRIDORS");
150     strcpy(labels[ CMD_CONNECT_CL] , "CONNECT TO CONTLOC");
151     strcpy(labels[ CMD_SEND_CL_GRID] , "SEND CONTLOC GRID");
152     strcpy(labels[ CMD_BUMP] , "BUMPER TEST");
153     strcpy(labels[ CMD_INIT_COMM] , "INIT ROBOT COMM");
154     strcpy(labels[ CMD_SEND_MSG] , "SEND MESSAGE");
155     strcpy(labels[ CMD_RECEIVE_MSG] , "RECEIVE MESSAGE");
156
157     strcpy(labels[ CMD_EXIT] , "EXIT");
158
159 //BEGIN SCOUT THESIS CHANGE
160     sprintf(Control_Panel, "Control [%d] Panel", r.id);
161
162     control_window.init_panel(CON_WIN_LEFT, CON_WIN_TOP,
163 CON_BUTTON_WIDTH,
164                         CON_BUTTON_HEIGHT, Control_Panel,
165                         CON_LAB_WIDTH, CON_LAB_HEIGHT, CON_NUM_CMD,
166                         CON_COLS, CON_ROWS, labels);
167     control_window.draw();
168
```

215

```
169        // Initialize evidence grid window
170
171        grid_window = new window(EGWIN_LEFT, EGWIN_TOP, EGWIN_RIGHT,
172    EGWIN_BOTTOM,
173                            "Evidence Grid");
174        grid_window->set_window(EGWIN_WC_LEFT, EGWIN_WC_BOTTOM,
175    EGWIN_WC_RIGHT,
176                            EGWIN_WC_TOP);
177        grid_window->iconify();
178
179        // Initialize navigation grid window
180
181        //    nav_window = new window(NAV_WIN_LEFT, NAV_WIN_TOP,
182    NAV_WIN_RIGHT,
183        //                        NAV_WIN_BOTTOM, "Navigation Grid");
184        //    nav_window->set_window(NAV_WIN_WC_LEFT, NAV_WIN_WC_BOTTOM,
185    NAV_WIN_WC_RIGHT,
186        //                        NAV_WIN_WC_TOP);
187        //    nav_window->iconify();
188
189        // Initialize global evidence grid window
190
191      sprintf(Global_Grid, "Global [%d] Grid", r.id);
192
193      global_window = new window(GLOBAL_WIN_LEFT, GLOBAL_WIN_TOP,
194                            GLOBAL_WIN_RIGHT, GLOBAL_WIN_BOTTOM,
195                            Global_Grid);
196    // END SCOUT THESIS CHANGE
197        global_window->set_window(GLOBAL_WIN_WC_LEFT, GLOBAL_WIN_WC_BOTTOM,
198                            GLOBAL_WIN_WC_RIGHT, GLOBAL_WIN_WC_TOP);
199        //    global_window->iconify();
200
201        // Initialize evidence grid sensor models
202
203    //    cout << "Evidence grid: <disabled>" << endl;
204
205        table_init();
206        model_init(&sonar_smd, &sonar_clear_smd);
207
208        // Initialize evidence grids
209
210        grid_init(&egrid, 0.0, 0.0);
211        grid_init(&old_grid, 0.0, 0.0);
212
213        grid_init_nav(&nav_grid, 0.0, 0.0);
214
215        grid_init_global(&global_grid, 0.0, 0.0);
216        grid_init_global(&old_global, 0.0, 0.0);
217        grid_init_global(&edge_grid, 0.0, 0.0);
218
219        // Initialize moving obstacles
220
221        for (i = 0; i < NUM_MOB; i++) {
222          mob_list[i].rand_init();
223        }
224
225        // Reset timers
226
```

```
227        timer = 0;
228
229        // Initialize file pointers
230
231        logfile = NULL;
232
233        // Turn on all sensors
234
235        r.sonar_on();
236        r.ir_on();
237        r.laser_on();
238
239        // Initialize cell count
240
241        cell_count = 0;
242
243        // BUMPER FIX INITIALIZATION
244
245        for (i = 0; i < NUM_TOUCH; i++) {
246          bumped[i] = 0;
247        }
248    }
249
250    /********** USER CONTROL FUNCTIONS **********/
251
252    void agent::control(void)
253    {
254        // Main control loop
255
256      int quit = 0;
257
258      do {
259        quit = user_command();
260      }
261      while (!quit);
262    }
263
264
265    void agent::power_check(void)
266    {
267      // Check battery power
268
269      char vostr[STRLEN];    // Voice string
270
271      gs();
272
273      cpu_volt = (double) (int) (voltCpuGet() * 100.0) / 100.0;
274      motor_volt = (double) (int) (voltMotorGet() * 100.0) / 100.0;
275
276      //  cout << "CPU voltage = " << cpu_volt << " : motor voltage = " <<
277    motor_volt
278      //    << endl;
279
280      //  cout << "CPU voltage = " << voltCpuGet() << " : motor voltage = "
281      //         << voltMotorGet() << endl;
282
283      if (cpu_volt < cpu_min) {
284        cpu_min = cpu_volt;
```

```
285         if (cpu_volt < CPU_DANGER_VOLTAGE) {
286           sprintf(vostr, "Danger, Danger: C P U voltage is %.2f.\n",
287   cpu_volt);
288           cout << vostr;
289           tk(vostr);
290         }
291         else if (cpu_volt < CPU_FULL_VOLTAGE) {
292           sprintf(vostr, "Warning: C P U voltage is %.2f.\n", cpu_volt);
293           cout << vostr;
294           tk(vostr);
295         }
296       }
297
298       if (motor_volt < motor_min) {
299         motor_min = motor_volt;
300         if (motor_volt < MOTOR_DANGER_VOLTAGE) {
301           sprintf(vostr, "Danger, Danger: Motor voltage is %.2f.\n",
302   motor_volt);
303           cout << vostr;
304           tk(vostr);
305         }
306         else if (motor_volt < MOTOR_FULL_VOLTAGE) {
307           sprintf(vostr, "Warning: Motor voltage is %.2f.\n", motor_volt);
308           cout << vostr;
309           tk(vostr);
310         }
311       }
312   }
313
314   int agent::user_command(void)
315   {
316     // Execute user command (if any)
317
318     int quit = 0;         // Set to 1 for exit command
319     int command;          // Command code
320
321     //  power_check();
322
323     control_window.refresh();
324     command = control_window.scan_panel();
325
326     switch(command) {
327     case CMD_EXPLORE:
328       exploration_lls();
329       break;
330     case CMD_NAV:
331       navigation();
332       break;
333     case CMD_NAV_KBD:
334       navigation_keyboard();
335       break;
336     case CMD_SAVE:
337       save_net();
338       break;
339     case CMD_LOAD:
340       load_net();
341       break;
342     case CMD_REDRAW:
```

```
343        pnet.display();
344        break;
345     case CMD_BUILD_GRID:
346        r.update();
347        grid_clear(egrid);
348        clear_robot(egrid, 0, 0);
349        sonar_sweep_seq(egrid);
350  //     laser_sweep_seq(egrid);
351        grid_display(grid_window, egrid);
352        break;
353     case CMD_SAVE_GRID:
354        save_grid(egrid);
355        break;
356     case CMD_LOAD_GRID:
357        load_grid(&egrid);
358        r.update();
359        grid_display(grid_window, egrid);
360        break;
361     case CMD_GRID:
362        r.update();
363        grid_display(grid_window, egrid);
364        break;
365     case CMD_CLEAR:
366        grid_copy(old_grid, egrid);
367        grid_clear(egrid);
368        grid_display(grid_window, egrid);
369        break;
370     case CMD_CLEAR_ROBOT:
371        grid_copy(old_grid, egrid);
372        r.update();
373        clear_robot(egrid, r.x, r.y);
374        grid_display(grid_window, egrid);
375        break;
376     case CMD_SONAR_SCAN:
377        grid_copy(old_grid, egrid);
378        r.update();
379  //  SCOUT THESIS CHANGE - in line below changed r.turret to r.theta
380        sonar_scan(sonar_smd, sonar_clear_smd, egrid, r.x, r.y, r.theta);
381        grid_display(grid_window, egrid);
382        break;
383     case CMD_SONAR_SWEEP:
384        grid_copy(old_grid, egrid);
385        clear_robot(egrid, 0, 0);
386        sonar_sweep_seq(egrid);
387        grid_display(grid_window, egrid);
388        break;
389     case CMD_SONAR_SWEEP_ABS:
390        grid_copy(old_grid, egrid);
391        r.update();
392        clear_robot(egrid, r.x, r.y);
393        sonar_sweep_abs_seq(egrid);
394        grid_display(grid_window, egrid);
395        break;
396     case CMD_CLEAR_SONAR:
397        grid_copy(old_grid, egrid);
398        grid_clear(egrid);
399        r.update();
400        clear_robot(egrid, 0, 0);
```

```
401         sonar_sweep_seq(egrid);
402         grid_display(grid_window, egrid);
403         break;
404     case CMD_LASER_SCAN:
405         grid_copy(old_grid, egrid);
406         r.update();
407         //  Replaced r.turret with r.theta in line below
408         laser_scan(egrid, r.x, r.y, r.theta);   // SCOUT THESIS change for
409     Scout with fixed body laser
410         grid_display(grid_window, egrid);
411         break;
412     case CMD_LASER_SWEEP:
413         grid_copy(old_grid, egrid);
414         r.update();
415         laser_sweep_seq(egrid);
416         grid_display(grid_window, egrid);
417         break;
418     case CMD_LASER_SWEEP_ABS:
419         grid_copy(old_grid, egrid);
420         r.update();
421         laser_sweep_abs_seq(egrid);
422         grid_display(grid_window, egrid);
423         break;
424     case CMD_CLEAR_LASER:
425         grid_copy(old_grid, egrid);
426         grid_clear(egrid);
427         r.update();
428         laser_sweep_seq(egrid);
429         grid_display(grid_window, egrid);
430         break;
431     case CMD_LLS_SCAN:
432         grid_copy(old_grid, egrid);
433         r.update();
434         lls_scan(sonar_smd, sonar_clear_smd, egrid, r.x, r.y, r.theta);   //
435     SCOUT THESIS  - see change above
436         grid_display(grid_window, egrid);
437         break;
438     case CMD_LLS_SWEEP:
439         grid_copy(old_grid, egrid);
440         r.update();
441         clear_robot(egrid, 0, 0);
442         lls_sweep_abs_seq(egrid);
443         grid_display(grid_window, egrid);
444         break;
445     case CMD_LLS_SWEEP_ABS:
446         grid_copy(old_global, global_grid);
447         r.update();
448         clear_robot(global_grid, 0, 0);
449         lls_sweep_seq(global_grid);
450         grid_display_global(global_grid);
451         break;
452     case CMD_CLEAR_LLS:
453         grid_copy(old_grid, egrid);
454         grid_clear(egrid);
455         r.update();
456         clear_robot(egrid, 0, 0);
457         lls_sweep_seq(egrid);
458         grid_display(grid_window, egrid);
```

```
459         break;
460     case CMD_GRID_UNDO:
461       grid_copy(egrid, old_grid);
462       grid_display(grid_window, egrid);
463       break;
464     case CMD_GRID_IDENT:
465       grid_ident_seq();
466       break;
467     case CMD_CENTER:
468       center_seq();
469       break;
470     case CMD_PLACE_MAP:
471       map_seq();
472       break;
473     case CMD_PLACE_IDENT:
474       ident_seq();
475       break;
476     case CMD_PLACE_GRID:
477       display_place_grid();
478       break;
479     case CMD_LOCAL_NAV:
480       local_navigation();
481       break;
482     case CMD_ADD_PLACE:
483       pnet.add_place();
484       break;
485     case CMD_EDIT_PLACE:
486       pnet.edit_place();
487       break;
488     case CMD_ADD_EDIT_LINK:
489       pnet.add_edit_link();
490       break;
491     case CMD_DELETE_LINK:
492       pnet.delete_link();
493       break;
494     case CMD_CLEAR_GLOBAL:
495       grid_copy(old_global, global_grid);
496       grid_clear(global_grid);
497       grid_display_global(global_grid);
498       num_front = 0;
499       num_visit = 0;
500       num_inac = 0;
501       break;
502     case CMD_SAVE_GLOBAL:
503       save_grid(global_grid);
504       break;
505     case CMD_LOAD_GLOBAL:
506       load_grid(&global_grid);
507       r.update();
508       grid_display_global(global_grid);
509       break;
510     case CMD_DISPLAY_GLOBAL:
511       grid_display_global(global_grid);
512       break;
513     case CMD_GLOBAL_UNDO:
514       grid_copy(global_grid, old_global);
515       grid_display_global(global_grid);
516       break;
```

```
517        case CMD_INTEGRATE_GRID:
518          integrate_grid(global_grid, egrid, (double) r.x / 120.0,
519                    (double) r.y / 120.0, (double) r.theta / 10.0);
520          grid_display_global(global_grid);
521          break;
522        case CMD_FIND_FRONTIERS:
523          find_frontiers();
524          break;
525        case CMD_DISPLAY_EDGES:
526          grid_display_global(global_grid);
527          grid_display_edges(region_map);
528          break;
529        case CMD_DISPLAY_FRONTIERS:
530          grid_display_global(global_grid);
531          grid_display_regions(region_map);
532          display_region_centroids(0.0, 0.0);
533          //     display_robot_region_centroids();
534          break;
535        case CMD_GOTO_FRONTIER:
536          frontier_navigate();
537          break;
538        case CMD_UPDATE_NAV_GRID:
539          update_nav_grid();
540          break;
541        case CMD_DETECT_CORRIDORS:
542          detect_corridors();
543          display_corridors();
544          break;
545        case CMD_CONNECT_CL:
546          connect_cl();
547          break;
548        case CMD_SEND_CL_GRID:
549          send_cl_grid();
550          break;
551        case CMD_BUMP:
552          bump_test();
553          break;
554        case CMD_INIT_COMM:
555          init_robot_comm();
556          break;
557        case CMD_SEND_MSG:
558          user_send_robot_message();
559          break;
560        case CMD_RECEIVE_MSG:
561          user_receive_robot_message();
562          break;
563        case CMD_EXIT:
564          terminate();
565          quit = 1;
566          break;
567      }
568      return(quit);
569  }
570
571  void agent::terminate(void)
572  {
573      // End session
574
```

```
575      int i;
576
577      // Delete mobstacles
578
579      for (i = 0; i < NUM_MOB; i++) {
580        mob_list[ i] .del_obs();
581      }
582
583      // Shut down robot
584
585      r.shutdown();
586   }
587
588   int agent::iscan(void)
589   {
590       // Scan for interrupt
591
592       int command;
593
594       control_window.refresh();
595       command = control_window.scan_panel();
596       if ((command == CMD_STOP) || (command == CMD_EXIT)) {
597         st();
598         return(ABORT);
599       }
600       else {
601         return(OK);
602       }
603   }
604
605   /********** BEHAVIOR CONTROL SYSTEMS **********/
606
607   void agent::bump_test(void)
608   {
609     grid_display_global(global_grid);
610
611     while(iscan() != ABORT) {
612        update();
613        bump_halt();
614     }
615   }
616
617   void agent::manual_exploration(void)
618   {
619     // Map territory under manual control
620
621     int net_status; // Place net changed status
622
623     timer = 0;
624     behavior_mode = EXPLORE_MODE;
625
626   // BEGIN SCOUT THESIS CHANGE
627     scout_vm(0, 0); // Necessary hack so robot will start moving later
628   // SCOUT THESIS CHANGE - changed pr to vm
629   // END SCOUT THESIS CHANGE
630
631     manual_exploration_seq();
632   }
```

```
633
634   void agent::exploration(void)
635   {
636      // Explore territory
637
638      behavior_mode = EXPLORE_MODE;
639
640      exploration_seq();
641   }
642
643   void agent::exploration_lls(void)
644   {
645      // Explore territory using laser-limited sonar
646
647      char comm_str[ STRLEN] ;                // Contloc communication string
648
649      behavior_mode = EXPLORE_LLS_MODE;
650
651      // Set relocalization interval
652
653      sprintf(comm_str, "reloc_distance = %d", EXPLORE_RELOC_DISTANCE);
654      cout << "comm str = <" << comm_str << ">" << endl;
655      write_comm(COMM_CHANNEL, comm_str, strlen(comm_str) + 1);
656
657      // Exploration sequence
658
659      exploration_lls_seq();
660   }
661
662   void agent::reactive_exploration(void)
663   {
664      // Explore territory reactively
665
666      int net_status; // Place net changed status
667   //   char logname[ STRLEN] ;        // Log file name
668   //   char apnname[ STRLEN] ;        // APN file name
669
670      timer = 0;
671      behavior_mode = EXPLORE_MODE;
672
673   /*   do {
674         cout << "Enter log file name ==> ";
675         cin >> logname;
676
677         logfile = new ofstream(logname);
678         if (logfile == NULL) {
679            cout << "Unable to open log file <" << logname << ">." << endl;
680         }
681      }
682      while(logfile == NULL);
683
684      cout << "Enter APN file name ==> ";
685      cin >> apnname;*/
686
687   //   reset();
688   //   pnet.clear_net();
689
690      update();
```

224

```
691     net_status = pnet.place_learn((double) r.x, (double) r.y,
692                          (double) r.theta / 10.0);
693     if (net_status & NEW_PLACE) {
694       map_seq();
695     }
696
697     reactive_exploration_seq();
698
699 //  logfile->close();
700 //  pnet.save(apnname);
701 }
702
703 void agent::multi_exploration(void)
704 {
705     // Explore territory (multiple trials)
706
707     char prefix[ STRLEN] ;  // Filename prefixes
708     char logname[ STRLEN] ; // Log filename
709     char apnname[ STRLEN] ; // APN filename
710     int trial_index;        // Trial index
711     int trial_start;        // Index for initial trial
712     int trial_end;  // Index for last trial
713     int rand_x, rand_y, rand_heading; // Random initial position
714
715     behavior_mode = EXPLORE_MODE;
716
717     cout << "Enter filename prefix ==> ";
718     cin >> prefix;
719
720     cout << "Enter starting trial number ==> ";
721     cin >> trial_start;
722
723     cout << "Enter ending trial number ==> ";
724     cin >> trial_end;
725
726     for(trial_index = trial_start; trial_index <= trial_end;
727 trial_index++) {
728         sprintf(logname, "%s%d.log", prefix, trial_index);
729         sprintf(apnname, "%s%d.apn", prefix, trial_index);
730
731         logfile = new ofstream(logname);
732         if (logfile == NULL) {
733           cout << "Unable to open log file <" << logname << ">." << endl;
734         }
735         else {
736           cout << "Opening log file <" << logname << ">." << endl;
737         }
738
739         reset();
740         pnet.clear_net();
741
742         // Set random initial position
743
744         rand_x = irand(PWIN_WC_LEFT + RAND_MARGIN, PWIN_WC_RIGHT -
745 RAND_MARGIN);
746         rand_y = irand(PWIN_WC_BOTTOM + RAND_MARGIN, PWIN_WC_TOP -
747 RAND_MARGIN);
748         rand_heading = irand(0, 3600);
```

225

```
749        place_robot(rand_x, rand_y, rand_heading, rand_heading);
750
751        // Hack to make sure robot isn't teleported into wall
752
753    // BEGIN SCOUT THESIS CHANGE
754        scout_vm(1, 0);    // TEMP FIX- changed pr to vm
755        scout_vm(-1, 0);    // TEMP FIX - changed pr to vm
756    // END SCOUT THESIS CHANGE
757
758        update();
759        pnet.place_learn((double) r.x, (double) r.y, (double) r.theta /
760    10.0);
761
762        exploration_seq();
763
764        if (logfile != NULL) {
765           logfile->close();
766        }
767        pnet.save(apnname);
768      }
769    }
770
771    void agent::multi_reactive_exploration(void)
772    {
773      // Explore territory reactively (multiple trials)
774
775      char prefix[ STRLEN] ;   // Filename prefixes
776      char logname[ STRLEN] ;  // Log filename
777      char apnname[ STRLEN] ;  // APN filename
778      int trial_index;        // Trial index
779      int trial_start;        // Index for initial trial
780      int trial_end;   // Index for last trial
781
782      behavior_mode = EXPLORE_MODE;
783
784      cout << "Enter filename prefix ==> ";
785      cin >> prefix;
786
787      cout << "Enter starting trial number ==> ";
788      cin >> trial_start;
789
790      cout << "Enter ending trial number ==> ";
791      cin >> trial_end;
792
793      for(trial_index = trial_start; trial_index <= trial_end;
794    trial_index++) {
795        sprintf(logname, "%s%d.log", prefix, trial_index);
796        sprintf(apnname, "%s%d.apn", prefix, trial_index);
797
798        logfile = new ofstream(logname);
799        if (logfile == NULL) {
800          cout << "Unable to open log file <" << logname << ">." << endl;
801        }
802        else {
803          cout << "Opening log file <" << logname << ">." << endl;
804        }
805
806        reset();
```

```
807        pnet.clear_net();

809        update();
810        pnet.place_learn((double) r.x, (double) r.y, (double) r.theta /
811    10.0);

813        reactive_exploration_seq();

815        if (logfile != NULL) {
816           logfile->close();
817        }
818        pnet.save(apnname);
819      }
820    }

822    void agent::navigation(void)
823    {
824        // Navigate to destination specified with mouse

826        char comm_str[ STRLEN] ;         // Contloc communication string
827        char vostr[ STRLEN] ;    // Voice string
828        double gx, gy;   // Destination point (world coords)

830        // Wait for user to click on destination in global window

832        grid_display_global(global_grid);

834        while(global_window->world_mouse(gx, gy) == 0);

836        sprintf(vostr, "Navigating to %d, %d.\n", (int) gx, (int) gy);
837        cout << vostr;
838        tk(vostr);

840        // Mark destination in window

842        global_window->set_color("red");
843        global_window->display_circle(gx, gy, CENTROID_MARK_RADIUS);
844        global_window->display_line(gx - CENTROID_MARK_RADIUS, gy,
845                           gx + CENTROID_MARK_RADIUS, gy);
846        global_window->display_line(gx, gy - CENTROID_MARK_RADIUS,
847                           gx, gy + CENTROID_MARK_RADIUS);
848        global_window->set_color("black");

850        // Set relocalization interval

852        sprintf(comm_str, "reloc_distance = %d", NAV_RELOC_DISTANCE);
853        cout << "comm str = <" << comm_str << ">" << endl;
854        write_comm(COMM_CHANNEL, comm_str, strlen(comm_str) + 1);

856        // Navigate to destination

858        refresh_all();
859        path_nav_seq(gx, gy);

861        r.move_to_xy((int) gx, (int) gy);
862        r.face_angle(0);

864        tk("");    // Sometimes garbage gets stuck in the voice buffer
```

```
865
866     sprintf(vostr, "Arrived at destination.\n");
867     cout << vostr;
868     tk(vostr);
869   }
870
871   void agent::navigation_keyboard(void)
872   {
873     // Navigate to destination specified with keyboard
874
875     char comm_str[STRLEN];       // Contloc communication string
876     char vostr[STRLEN];    // Voice string
877     double gx, gy;   // Destination point (world coords)
878     double gtheta;   // Destination orientation
879
880     // Ask user to enter destination
881
882     cout << "Enter destination (x, y, theta) (1/10 in, 1/10 deg) ==> ";
883     cin >> gx >> gy >> gtheta;
884
885     sprintf(vostr, "Navigating to %d, %d (%d).\n", (int) gx, (int) gy,
886             (int) gtheta);
887     cout << vostr;
888     tk(vostr);
889
890     // Mark destination in window
891
892     grid_display_global(global_grid);
893
894     global_window->set_color("red");
895     global_window->display_circle(gx, gy, CENTROID_MARK_RADIUS);
896     global_window->display_line(gx - CENTROID_MARK_RADIUS, gy,
897                             gx + CENTROID_MARK_RADIUS, gy);
898     global_window->display_line(gx, gy - CENTROID_MARK_RADIUS,
899                             gx, gy + CENTROID_MARK_RADIUS);
900     global_window->set_color("black");
901
902     // Set relocalization interval
903
904     sprintf(comm_str, "reloc_distance = %d", NAV_RELOC_DISTANCE);
905     cout << "comm str = <" << comm_str << ">" << endl;
906     write_comm(COMM_CHANNEL, comm_str, strlen(comm_str) + 1);
907
908     // Navigate to destination
909
910     refresh_all();
911     path_nav_seq(gx, gy);
912
913     r.move_to_xy((int) gx, (int) gy);
914     r.face_angle((int) gtheta);
915
916     tk("");    // Sometimes garbage gets stuck in the voice buffer
917
918     sprintf(vostr, "Arrived at destination.\n");
919     cout << vostr;
920     tk(vostr);
921   }
922
```

```
923    void agent::local_navigation(void)
924    {
925        // Navigate to local coordinate point
926
927        int x, y;                    // Local destination coordinates
928
929        cout << "Enter destination point (x, y) ==> ";
930        cin >> x >> y;
931
932        local_nav_seq(x, y);
933    }
934
935    void agent::frontier_navigate(void)
936    {
937      // Navigate to frontier centroid
938
939      int front_index;       // Index of destination frontier
940
941      if (num_front == 0) {
942        cout << "No frontiers detected." << endl;
943        return;
944      }
945
946      do {
947        cout << "Enter frontier index ==> ";
948        cin >> front_index;
949        if ((front_index < 0) || (front_index >= num_front)) {
950          cout << "Unknown frontier -- must be in range [ 0.." << num_front
951    << "] ."
952              << endl;
953        }
954      }
955      while((front_index < 0) || (front_index >= num_front));
956
957      frontier_nav_seq(front_index);
958    }
959
960    /********** BEHAVIORAL SEQUENCERS **********/
961
962    void agent::manual_exploration_seq(void)
963    {
964      // Manual exploration sequencer
965
966      int net_status; // Place net changed status
967
968      cout << "Exploring under manual control..." << endl;
969
970      do {
971        update();
972        net_status = pnet.place_learn((double) r.x, (double) r.y,
973                              (double) r.theta / 10.0);
974
975        if (net_status & NEW_PLACE) {
976          cout << "Stop." << endl;
977          tk("Stop.");
978          st();
979          ws(1, 1, 1, 5);
980
```

```
981        map_seq();
982      }
983    }
984    while(iscan() != ABORT);
985
986    cout << "Exploration complete." << endl;
987  }
988
989  void agent::exploration_seq(void)
990  {
991    // Exploration sequencer
992
993    int front_index = 0;   // Frontier destination index
994    int nav_status = OK;   // Navigation status
995
996    cout << "Exploring..." << endl;
997    tk("Exploring.");
998
999    update();
1000   clear_robot(global_grid, r.x, r.y);
1001   sonar_sweep_abs_seq(global_grid);
1002
1003   find_frontiers();
1004
1005   while((num_front > 0) && (nav_status != ABORT) && (front_index != -1))
1006   {
1007     front_index = closest_frontier((double) r.x, (double) r.y);
1008
1009     if (front_index != -1) {
1010       nav_status = frontier_nav_seq(front_index);
1011       //     clear_robot(global_grid, r.x, r.y);
1012       //     sonar_sweep_seq(global_grid);
1013       find_frontiers();
1014     }
1015   }
1016
1017   cout << "Exploration complete." << endl;
1018   tk("Exploration complete.");
1019 }
1020
1021 void agent::exploration_lls_seq(void)
1022 {
1023   // Exploration sequencer using laser-limited sonar
1024
1025
1026   char local_filename [ STRLEN];            // Filename for local grid
1027   char local_posinfo [ STRLEN];         // Position info for local grid
1028 file
1029   char global_filename[ STRLEN];      // Filename for global grid
1030   char global_posinfo[ STRLEN];       // Position info for global grid
1031 file
1032   char message[ STRLEN];         // Message for multirobot communications
1033
1034   double tx = 0.0, ty = 0.0;         // Registration translation vector
1035   double ttheta = 0.0;               // Registration rotation
1036   double score;                      // Registration score for local
1037 grid
1038
```

230

```
1039      int front_index = 0;    // Frontier destination index
1040      int nav_status = OK;    // Navigation status
1041      int occ;             // Number of occupied cells in global grid
1042      int unocc;                  // Number of unoccupied cells in global grid
1043
1044      cout << "Exploring..." << endl;
1045      tk("Exploring.");
1046
1047
1048
1049    // NEW SCOUT THESIS CHANGE below
1050    // If robot is robot 1 it is the SERVER robot and will send its global
1051    map out to
1052    // the other CLIENT robots
1053    // if robot is not number 1 then it is a CLIENT robot and will only
1054    write its
1055    // local scan to file
1056    // in this way I hope to slow down error buildup in the map
1057
1058     if (r.id == 1)  {
1059
1060       sprintf(global_filename, "global%d.eg", r.id);
1061
1062       // Sweep from initial position and find frontiers
1063
1064       update();
1065       clear_robot(global_grid, r.x, r.y);
1066
1067    // BEGIN SCOUT THESIS CHANGE
1068    // instead of using laser limited sonar use just the sonars
1069
1070    //  lls_sweep_abs_seq(global_grid);          //  commented out for Scout
1071       sonar_sweep_abs_seq(global_grid);    // use sonars only to explore
1072    // END SCOUT THESIS CHANGE
1073
1074       //  grid_display(grid_window, egrid);
1075
1076       // Save global grid
1077
1078       sprintf(global_posinfo, "%d %d %d", 0, 0, 0);
1079       save_grid_file(global_grid, global_filename, global_posinfo);
1080
1081       // Notify other robot
1082
1083       if (multi_mode) {
1084         send_robot_message(global_filename);
1085       }
1086
1087       // Display global grid
1088
1089       grid_display_global(global_grid);
1090
1091       // Send grid to continuous localization
1092
1093       grid_count_occ(global_grid, &occ, &unocc);
1094       cout << "Global grid cells: mapped = " << occ + unocc
1095            << " : occupied  = " << occ << endl;
1096       if (occ >= CONTLOC_MIN_OCC) {
```

231

```
1097        send_cl_grid();
1098      }
1099
1100      // Check for new map from other robot
1101
1102      if (multi_mode) {
1103        integrate_remote_map();
1104      }
1105
1106      // Find initial frontiers
1107
1108      find_frontiers();
1109
1110      while(nav_status != ABORT) {
1111        if (num_front > 0) {
1112          // Navigate to closest frontier (index = -1 if inaccessible or
1113  visited)
1114
1115          front_index = closest_frontier((double) r.x, (double) r.y);
1116          if (front_index != -1) {
1117          nav_status = frontier_nav_seq(front_index);
1118          } .
1119        }
1120
1121        if ((num_front == 0) || (front_index == -1)) {
1122          if (iscan() == ABORT)  {         // add check for interrupts from
1123  control panel
1124            nav_status = ABORT;
1125          }
1126          else {
1127            cout << "No frontiers remaining, sweeping sensors..." << endl;
1128            tk("No frontiers, sweeping.");
1129            nav_status = NO_FRONTIERS;
1130          }
1131        }
1132
1133        if ((nav_status != ABORT) && (nav_status != NO_PATH)) {
1134          clear_robot(global_grid, r.x, r.y);
1135
1136  // BEGIN SCOUT THESIS CHANGE
1137  // instead of using laser limited sonar use just the sonars
1138
1139  //      lls_sweep_abs_seq(global_grid);   // commented out for Scout
1140          sonar_sweep_abs_seq(global_grid);
1141  // END SCOUT THESIS CHANGE
1142
1143          //    grid_display(grid_window, egrid);
1144
1145          // Save global grid
1146
1147          sprintf(global_posinfo, "%d %d %d", 0, 0, 0);
1148          save_grid_file(global_grid, global_filename, global_posinfo);
1149
1150          // Notify other robot
1151
1152          if (multi_mode) {
1153          send_robot_message(global_filename);
1154          }
```

```
1155
1156            // Display global grid
1157
1158            grid_display_global(global_grid);
1159
1160            // Send grid to continuous localization
1161
1162            grid_count_occ(global_grid, &occ, &unocc);
1163
1164            cout << "Global grid cells: mapped = " << occ + unocc
1165                << " : occupied  = " << occ << endl;
1166
1167            if (occ >= CONTLOC_MIN_OCC) {
1168            send_cl_grid();
1169            }
1170
1171            // Check for new map from other robot
1172
1173            if (multi_mode) {
1174            integrate_remote_map();
1175            }
1176
1177            // Find new frontiers
1178
1179            find_frontiers();
1180        }
1181     }
1182
1183     }    // close for if r.id==1
1184
1185
1186
1187  // NEW MAJOR SCOUT THESIS change
1188  // now handle the case of the CLIENT robots that just write their
1189  // local maps
1190
1191   else {                  // r.id != 1
1192
1193     sprintf(local_filename, "local%d.eg", r.id);
1194
1195     // Sweep from initial position and find frontiers
1196
1197     update();
1198     grid_clear(egrid);        // clear the old local grid prior to scanning
1199     clear_robot(egrid, 0, 0);   // mark the cells under the robot as
1200  unoccupied
1201  //   clear_robot(global_grid, r.x, r.y);
1202
1203  // BEGIN SCOUT THESIS CHANGE
1204  // instead of using laser limited sonar use just the sonars
1205
1206  //   lls_sweep_abs_seq(global_grid);          //  commented out for Scout
1207  //   sonar_sweep_abs_seq(global_grid);   // use sonars only to explore in
1208  global position
1209     sonar_sweep_seq(egrid);      // use sonars only to make local scan
1210  centered around robot
1211  // END SCOUT THESIS CHANGE
1212
```

```
1213        //  grid_display(grid_window, egrid);
1214
1215        // Register local grid with global grid - necessary when using robot
1216    base position
1217        // for scanning vice global position
1218
1219        tx = (double) r.x / 120.0;
1220        ty = (double) r.y / 120.0;
1221        ttheta = 0.0;
1222
1223        // Save local grid
1224
1225        sprintf(local_posinfo, "%d %d %d", r.x, r.y, 0);
1226        save_grid_file(egrid, local_filename, local_posinfo);
1227
1228        // Notify other robot
1229
1230        if (multi_mode) {
1231            send_robot_message(local_filename);
1232        }
1233
1234    //  Integrate local grid with global grid
1235        integrate_grid(global_grid, egrid, tx, ty, ttheta);
1236
1237
1238        // Display global grid
1239        grid_display_global(global_grid);
1240
1241        // Send grid to continuous localization
1242
1243        grid_count_occ(global_grid, &occ, &unocc);
1244        cout << "Global grid cells: mapped = " << occ + unocc
1245            << " : occupied  = " << occ << endl;
1246        if (occ >= CONTLOC_MIN_OCC) {
1247            send_cl_grid();
1248        }
1249
1250        // Check for new map from other robot
1251
1252        if (multi_mode) {
1253            integrate_remote_map();
1254        }
1255
1256        // Find initial frontiers
1257
1258        find_frontiers();
1259
1260        while(nav_status != ABORT) {
1261            if (num_front > 0) {
1262                // Navigate to closest frontier (index = -1 if inaccessible or
1263    visited)
1264
1265                front_index = closest_frontier((double) r.x, (double) r.y);
1266                if (front_index != -1) {
1267                nav_status = frontier_nav_seq(front_index);
1268                }
1269            }
1270
```

```
1271        if ((num_front == 0) || (front_index == -1)) {
1272          if (iscan() == ABORT)  {              // check for interrupts from
1273    control panel
1274            nav_status = ABORT;
1275          }
1276          else  {
1277            cout << "No frontiers remaining, sweeping sensors..." << endl;
1278            tk("No frontiers, sweeping.");
1279            nav_status = NO_FRONTIERS;
1280          }
1281        }
1282
1283        if ((nav_status != ABORT) && (nav_status != NO_PATH)) {
1284          grid_clear(egrid);
1285          clear_robot(egrid, 0, 0);
1286
1287   // BEGIN SCOUT THESIS CHANGE
1288   // instead of using laser limited sonar use just the sonars
1289
1290   //      lls_sweep_abs_seq(global_grid);    // commented out for Scout
1291   //       sonar_sweep_abs_seq(global_grid);
1292          sonar_sweep_seq(egrid);
1293   // END SCOUT THESIS CHANGE
1294
1295          //    grid_display(grid_window, egrid);
1296
1297     // Register local grid with global grid - necessary when using robot
1298   base position
1299     // for scanning vice global position
1300
1301        tx = (double) r.x / 120.0;
1302        ty = (double) r.y / 120.0;
1303        ttheta = 0.0;
1304
1305
1306
1307        // Save local grid
1308
1309        sprintf(local_posinfo, "%d %d %d", r.x, r.y, 0);
1310        save_grid_file(egrid, local_filename, local_posinfo);
1311
1312        // Notify other robot
1313
1314        if (multi_mode) {
1315        send_robot_message(local_filename);
1316        }
1317
1318     // Integrate local grid with global grid
1319        integrate_grid(global_grid, egrid, tx, ty, ttheta);
1320
1321        // Display global grid
1322
1323        grid_display_global(global_grid);
1324
1325        // Send grid to continuous localization
1326
1327        grid_count_occ(global_grid, &occ, &unocc);
1328
```

```
1329            cout << "Global grid cells: mapped = " << occ + unocc
1330               << " : occupied  = " << occ << endl;
1331
1332            if (occ >= CONTLOC_MIN_OCC) {
1333            send_cl_grid();
1334            }
1335
1336            // Check for new map from other robot
1337
1338            if (multi_mode) {
1339            integrate_remote_map();
1340            }
1341
1342            // Find new frontiers
1343
1344            find_frontiers();
1345          }
1346      }
1347
1348    }   // close for else r.id != 1
1349
1350  //   END NEW MAJOR THESIS change
1351
1352
1353
1354      cout << "Exploration complete." << endl;
1355      tk("Exploration complete.");
1356  }
1357
1358
1359
1360  void agent::reactive_exploration_seq(void)
1361  {
1362      // Reactive exploration sequencer
1363
1364      cout << "Exploring reactively..." << endl;
1365      tk("Exploring");
1366
1367      do {
1368        update();
1369
1370        set_defaults();
1371        if (reactive_explore_behaviors() == 0) {
1372          execute();
1373        }
1374      }
1375      while((iscan() != ABORT) && (timer <= TRIAL_LENGTH));
1376
1377      cout << "Exploration complete." << endl;
1378  }
1379
1380  int agent::navigation_seq(void)
1381  {
1382          // Follow path to destination
1383          // (returns ABORT if interrupt or error, OK otherwise)
1384
1385          char vostr[ STRLEN] ;        // Voice string
1386          int suc[ PLACE_UNITS] ;      // Succesor list
```

236

```
1387        int gx, gy;                  // Gateway location
1388        int arrived = 0;             // 1 when arrived at destination, 0
1389   otherwise
1390        int nav_status = OK;         // Navigation status
1391        int i;
1392
1393        behavior_mode = NAVIGATION_MODE;
1394
1395        cout << "Navigating to place [" << destin << "]." << endl;
1396        sprintf(vostr, "Navigating to place %d.\n", destin);
1397        tk(vostr);
1398
1399        //    ident_seq();
1400
1401        //    cout << "Enter current place index ==> ";
1402        //    cin >> pnet.windex;
1403
1404        while((pnet.windex != destin) && (nav_status != ABORT)) {
1405          pnet.find_paths(destin, suc);
1406          pnet.display();
1407
1408          cout << endl << "Place transition list:" << endl;
1409          for (i = 0; i < pnet.num_units; i++) {
1410              cout << "[" << i << "] --> [" << suc[i] << "]" << endl;
1411          }
1412          cout << endl;
1413
1414          if (suc[pnet.windex] == -1) {
1415              cout << "navigate_seq: No way to get from place [" <<
1416   pnet.windex
1417                   << "] to place [" << destin << "]." << endl;
1418              return(ABORT);
1419          }
1420
1421          if (pnet.link[pnet.windex][suc[pnet.windex]] == NULL) {
1422              cout << "navigate_seq: Nonexistent link [" << pnet.windex
1423                 << "] --> [" << suc[pnet.windex] << "]." << endl;
1424              return(ABORT);
1425          }
1426
1427          gx = pnet.link[pnet.windex][suc[pnet.windex]]->gateway_x;
1428          gy = pnet.link[pnet.windex][suc[pnet.windex]]->gateway_y;
1429
1430          cout << "Navigating to [" << pnet.windex << "] --> ["
1431               << suc[pnet.windex] << "] gateway at (" << gx << ", " << gy
1432   << ")."
1433                   << endl;
1434
1435          nav_status = local_nav_seq(gx, gy);
1436
1437          if (nav_status != ABORT) {
1438            ident_seq();
1439          }
1440        }
1441
1442        if (nav_status == ABORT) {
1443          cout << "Aborted." << endl;
1444        }
```

```
1445        else {
1446            cout << "Arrived at destination place [" << destin << "]." <<
1447    endl;
1448        }
1449        return(nav_status);
1450    }
1451
1452    int agent::local_nav_seq(int gx, int gy)       // Local destination
1453    coordinates
1454    {
1455        // Local navigation sequencer
1456
1457        char vostr[ STRLEN] ;            // Voice string
1458        double dist;                     // Distance from goal
1459        double min_dist;                 // Minimum distance to goal so far
1460        double bearing;            // Bearing to goal
1461        int nav_status = 0;              // 1: arrived, 0: otherwise
1462        int stall_count = 0;             // Timesteps since progress made toward
1463    goal
1464
1465        update();
1466
1467        dist = hypot((double) (gx - r.x), (double) (gy - r.y)) / 10.0;
1468        //   cout << "Distance from goal = " << dist << " inches" << endl;
1469
1470        bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;
1471        //   cout << "Bearing to goal = " << bearing << endl;
1472
1473        min_dist = dist;
1474
1475        sprintf(vostr, "Navigating to %d %d.\n", gx, gy);
1476        //   tk(vostr);
1477        cout << vostr;
1478
1479        if ((iscan() != ABORT) && (dist > LOCAL_NAV_TOLERANCE)) {
1480            r.face_angle_fast((int) (bearing * 10.0));
1481        }
1482
1483        while((iscan() != ABORT) && (dist > LOCAL_NAV_TOLERANCE) &&
1484            (stall_count < STALL_TIMEOUT)) {
1485
1486            update();
1487
1488            bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;
1489            if (angle_diff(bearing, (double) r.theta / 10.0) > LOCAL_TIP_ANGLE)
1490    {
1491                r.face_angle_fast((int) (bearing * 10.0));
1492            }
1493            else {
1494                set_defaults();
1495                nav_status = local_navigation_behaviors(gx, gy);
1496                execute();
1497            }
1498
1499            dist = hypot((double) (gx - r.x), (double) (gy - r.y)) / 10.0;
1500            //     cout << "Distance from goal = " << dist << "  inches" << endl;
1501
1502            if (dist < min_dist) {
```

238

```
1503        min_dist = dist;
1504        stall_count = 0;
1505      }
1506      else {
1507        stall_count++;
1508        if (stall_count % 5 == 0) {
1509        sprintf(vostr, "Stalled for %d steps.\n", stall_count);
1510        cout << vostr;
1511        tk(vostr);
1512        }
1513      }
1514    }
1515
1516    st();
1517
1518    if (stall_count >= STALL_TIMEOUT) {
1519      sprintf(vostr, "Navigation timeout.\n",
1520            stall_count);
1521      cout << vostr;
1522      tk(vostr);
1523      return(TIMEOUT);
1524    }
1525    else if (dist > LOCAL_NAV_TOLERANCE) {
1526      cout << "Aborted." << endl;
1527      tk("Aborted.");
1528      return(ABORT);
1529    }
1530
1531    cout << "Arrived." << endl;
1532    //  tk("Arrived.");
1533    return(OK);
1534 }
1535
1536 int agent::path_local_nav_seq(path p,          // Path to folloq
1537                               int &waypoint)   // Index of next waypoint
1538 {
1539    // Local navigation sequencer for path following
1540
1541    char message[ STRLEN] ;       // Message from other robot
1542    char vostr[ STRLEN] ;         // Voice string
1543    double dist;                  // Distance from goal
1544    double min_dist;              // Minimum distance to goal so far
1545    double close_dist;            // Distance to closest waypoint
1546    double bearing;         // Bearing to goal
1547    int gx, gy;                   // Waypoint coordinates
1548    int nav_status = 0;           // 1: arrived, 0: otherwise
1549    int stall_count = 0;          // Timesteps since progress made toward
1550 goal
1551    int close_index = waypoint; // Index of closest waypoint
1552    int i;
1553
1554    // Set goal to next waypoint
1555
1556    gx = p.x[ waypoint] ;
1557    gy = p.y[ waypoint] ;
1558
1559    // Update robot state
1560
```

```
1561      update();
1562
1563      // Find distance/bearing to goal
1564
1565      dist = hypot((double) (gx - r.x), (double) (gy - r.y)) / 10.0;
1566      //   cout << "Distance from goal = " << dist << " inches" << endl;
1567
1568      bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;
1569      //   cout << "Bearing to goal = " << bearing << endl;
1570
1571      min_dist = dist;
1572
1573      sprintf(vostr, "Navigating to %d %d.\n", gx, gy);
1574      //   tk(vostr);
1575      cout << vostr;
1576
1577      // Find distance to closest waypoint
1578
1579      close_dist = closest_waypoint(p, r.x, r.y, waypoint, close_index);
1580
1581      while((iscan() != ABORT) && (close_dist > LOCAL_NAV_TOLERANCE) &&
1582           (stall_count < STALL_TIMEOUT)) {
1583
1584        // Update robot state
1585
1586        update();
1587
1588        // Stop if collision
1589
1590        bump_halt();
1591
1592        // Realign if turret is misaligned with base
1593
1594        maintain_alignment();
1595
1596        // Find bearing to goal
1597
1598        bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;
1599
1600        cout << "goal [" << waypoint << "] : bearing = " << bearing
1601           << " : dist = " << dist << " | closest [" << close_index
1602           << "] : dist = " << close_dist << endl;
1603
1604        // Orient toward open corridor and advance
1605
1606        goal_corridor_orient(gx, gy);
1607        corridor_advance();
1608
1609        // Check distance from goal
1610
1611        dist = hypot((double) (gx - r.x), (double) (gy - r.y)) / 10.0;
1612
1613        if (dist < min_dist) {
1614
1615          // If progress has been made, reset stall counter
1616
1617          min_dist = dist;
1618          stall_count = 0;
```

240

```cpp
1619        }
1620        else {
1621
1622          // Otherwise, increment stall counter
1623
1624          stall_count++;
1625          if (stall_count % 5 == 0) {
1626          sprintf(vostr, "Stalled for %d steps.\n", stall_count);
1627          cout << vostr;
1628          tk(vostr);
1629          }
1630        }
1631
1632        // Find distance to closest waypoint
1633
1634        close_dist = closest_waypoint(p, r.x, r.y, waypoint, close_index);
1635      }
1636
1637      // Determine why navigation terminated
1638
1639      if (stall_count >= STALL_TIMEOUT) {                  // Timeout
1640        sprintf(vostr, "Navigation timeout.\n",
1641              stall_count);
1642        cout << vostr;
1643        tk(vostr);
1644        return(TIMEOUT);
1645      }
1646      else if (close_dist > LOCAL_NAV_TOLERANCE) {         // User abort
1647        cout << "Aborted." << endl;
1648        tk("Aborted.");
1649        return(ABORT);
1650      }
1651
1652      cout << "Arrived." << endl;                    // Success
1653      //  tk("Arrived.");
1654
1655      // Advance to next waypoint on path after closest waypoint
1656
1657      waypoint = close_index + 1;
1658
1659      return(OK);
1660    }
1661
1662    int agent::local_cont_nav_seq(int gx, int gy)    // Local destination
1663    coords
1664    {
1665      // Local navigation sequencer (continuous motion)
1666
1667      char vostr[ STRLEN] ;          // Voice string
1668      double dist;                   // Distance from goal
1669      double min_dist;               // Minimum distance to goal so far
1670      double bearing;          // Bearing to goal
1671      int nav_status = 0;            // 1: arrived, 0: otherwise
1672      int stall_count = 0;           // Timesteps since progress made toward
1673    goal
1674
1675      update();
1676
```

```
1677      dist = hypot((double) (gx - r.x), (double) (gy - r.y)) / 10.0;
1678      //  cout << "Distance from goal = " << dist << " inches" << endl;
1679
1680      bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;
1681      //  cout << "Bearing to goal = " << bearing << endl;
1682
1683      min_dist = dist;
1684
1685      sprintf(vostr, "Navigating to %d %d.\n", gx, gy);
1686      //  tk(vostr);
1687      cout << vostr;
1688
1689      if (angle_diff(bearing, (double) r.theta / 10.0) > LOCAL_TIP_ANGLE) {
1690        r.face_angle_fast((int) (bearing * 10.0));
1691      }
1692
1693      while((iscan() != ABORT) && (dist > LOCAL_NAV_TOLERANCE) &&
1694           (stall_count < STALL_TIMEOUT)) {
1695
1696        update();
1697
1698        bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;
1699        if (angle_diff(bearing, (double) r.theta / 10.0) > LOCAL_TIP_ANGLE)
1700    {
1701          r.face_angle_fast((int) (bearing * 10.0));
1702        }
1703        else {
1704          set_defaults();
1705          nav_status = local_navigation_behaviors(gx, gy);
1706          execute();
1707        }
1708
1709        dist = hypot((double) (gx - r.x), (double) (gy - r.y)) / 10.0;
1710        //    cout << "Distance from goal = " << dist << " inches" << endl;
1711
1712        if (dist < min_dist) {
1713          min_dist = dist;
1714          stall_count = 0;
1715        }
1716        else {
1717          stall_count++;
1718          if (stall_count % 5 == 0) {
1719          sprintf(vostr, "Stalled for %d steps.\n", stall_count);
1720          cout << vostr;
1721          tk(vostr);
1722          }
1723        }
1724      }
1725
1726      //  st();
1727
1728      if (stall_count >= STALL_TIMEOUT) {
1729        sprintf(vostr, "Navigation timeout.\n",
1730              stall_count);
1731        cout << vostr;
1732        tk(vostr);
1733        return(TIMEOUT);
1734      }
```

```
1735      else if (dist > LOCAL_NAV_TOLERANCE) {
1736        cout << "Aborted." << endl;
1737        tk("Aborted.");
1738        return(ABORT);
1739      }
1740
1741      cout << "Arrived." << endl;
1742      //  tk("Arrived.");
1743      return(OK);
1744    }
1745
1746    int agent::local_nav_seq_alt(int gx, int gy,     // Goal coordinates
1747                             int ax, int ay)     // Alternate goal coordinates
1748    {
1749      // Local navigation sequencer (with alternate goal)
1750
1751      char vostr[ STRLEN] ;           // Voice string
1752      double dist;                    // Distance from goal
1753      double alt_dist;                // Distance from alternate goal
1754      double min_dist;                // Minimum distance to goal so far
1755      double bearing;          // Bearing to goal
1756      int nav_status = 0;             // 1: arrived, 0: otherwise
1757      int stall_count = 0;            // Timesteps since progress made toward
1758    goal
1759      int interrupt;          // Interrupt code
1760
1761      update();
1762
1763      dist = hypot((double) (gx - r.x), (double) (gy - r.y)) / 10.0;
1764      //  cout << "Distance from goal = " << dist << " inches" << endl;
1765
1766      alt_dist = hypot((double) (ax - r.x), (double) (ay - r.y)) / 10.0;
1767      //    cout << "Distance from alternate goal = " << alt_dist << "
1768    inches"
1769      //    << endl;
1770
1771      bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;
1772      //  cout << "Bearing to goal = " << bearing << endl;
1773
1774      min_dist = dist;
1775
1776      sprintf(vostr, "Navigating to %d %d.\n", gx, gy);
1777      //  tk(vostr);
1778      cout << vostr;
1779
1780      cout << "Alternate goal: (" << ax << ", " << ay << ")" << endl;
1781
1782      r.face_angle_fast((int) (bearing * 10.0));
1783
1784      while(((interrupt = iscan()) != ABORT) && (dist > LOCAL_NAV_TOLERANCE)
1785    &&
1786          (stall_count < STALL_TIMEOUT) && (alt_dist > LOCAL_NAV_TOLERANCE))
1787    {
1788
1789        update();
1790
1791        bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;
```

```
1792        if (angle_diff(bearing, (double) r.theta / 10.0) > LOCAL_TIP_ANGLE)
1793    {
1794            r.face_angle_fast((int) (bearing * 10.0));
1795        }
1796        else {
1797            set_defaults();
1798            nav_status = local_navigation_behaviors(gx, gy);
1799            execute();
1800        }
1801
1802        dist = hypot((double) (gx - r.x), (double) (gy - r.y)) / 10.0;
1803        //    cout << "Distance from goal = " << dist << " inches" << endl;
1804
1805        alt_dist = hypot((double) (ax - r.x), (double) (ay - r.y)) / 10.0;
1806        //    cout << "Distance from alternate goal = " << alt_dist << "
1807    inches"
1808        //        << endl;
1809
1810        if (dist < min_dist) {
1811            min_dist = dist;
1812            stall_count = 0;
1813        }
1814        else {
1815            stall_count++;
1816            if (stall_count % 5 == 0) {
1817            sprintf(vostr, "Stalled for %d steps.\n", stall_count);
1818            cout << vostr;
1819            tk(vostr);
1820            }
1821        }
1822    }
1823
1824    st();
1825
1826    if (stall_count >= STALL_TIMEOUT) {
1827        sprintf(vostr, "Navigation timeout.\n",
1828                stall_count);
1829        cout << vostr;
1830        tk(vostr);
1831        return(TIMEOUT);
1832    }
1833
1834    if (interrupt == ABORT) {
1835        cout << "Aborted." << endl;
1836        tk("Aborted.");
1837        return(ABORT);
1838    }
1839
1840    if (dist <= LOCAL_NAV_TOLERANCE) {
1841        cout << "Arrived." << endl;
1842        //    tk("Arrived.");
1843        return(OK);
1844    }
1845
1846    if (alt_dist <= LOCAL_NAV_TOLERANCE) {
1847        cout << "Arrived at alternate goal." << endl;
1848        //    tk("Arrived at alternate goal.");
1849        return(ALT);
```

```
1850        }
1851
1852        cout << "local_nav_seq_alt: Illegal termination." << endl;
1853        exit(-1);
1854    }
1855
1856    void agent::center_seq(void)
1857    {
1858        // Move to center of current place
1859
1860        int cx, cy;          // Place center
1861        int ctheta;          // Place orientation
1862
1863        if (pnet.windex == -1) {
1864          cout << "Unable to center at unknown location." << endl;
1865          return;
1866        }
1867
1868        cx = (int) pnet.unit[ pnet.windex] .x;
1869        cy = (int) pnet.unit[ pnet.windex] .y;
1870        ctheta = (int) (pnet.unit[ pnet.windex] .theta * 10.0);
1871
1872    /*      cx = 0;
1873        cy = 0;
1874        ctheta = 0;*/
1875
1876        r.move_to_xy(cx, cy);
1877        r.face_angle(ctheta);
1878    // BEGIN SCOUT THESIS CHANGE
1879    // comment out call for turret alignment - is not necessary for SCOUT
1880    //      r.turret_align();
1881    // END SCOUT THESIS CHANGE
1882    }
1883
1884    int agent::path_nav_seq(double gx, double gy)    // World coords of goal
1885    {
1886      // Navigate to goal by planning and following path
1887
1888      path nav_path;                          // Navigation path
1889      int nav_status;                         // Navigation status
1890      int path_found;                         // 1 if path found, 0 otherwise
1891      int next_lls_point = NAV_LLS_SWEEP_INTERVAL;   // Waypoint for next LLS
1892    sweep
1893      int i, j;
1894
1895      path_found = path_plan(gx, gy, nav_path);
1896      if (!path_found) {
1897        return(NO_PATH);
1898      }
1899
1900      //  cout << "Press <enter> to continue." << endl;
1901      //  cin.get();
1902
1903      for (i = 1; i < nav_path.length; ) {
1904        nav_status = path_local_nav_seq(nav_path, i);
1905
1906        // Stop immediately at end of path
1907        // (so the robot doesn't crash if the goal is next to a wall)
```

```
1908        if (i == nav_path.length) {
1909          st();
1910          cout << "Stopping at path's end." << endl;
1911        }
1912
1913        if (i >= next_lls_point) {
1914          // Sweep laser at intervals (for contloc)
1915          lls_sweep_seq(egrid);
1916
1917          next_lls_point += NAV_LLS_SWEEP_INTERVAL;
1918        }
1919
1920        if (nav_status == ABORT) {                // User aborted
1921          return(ABORT);
1922        }
1923
1924        if (nav_status == ALT) {          // Arrived unexpectedly at goal
1925          display_path(nav_path, TRAV_PATH_COLOR, global_window);
1926          return(OK);
1927        }
1928
1929        if (nav_status == TIMEOUT) {      // Navigation timeout
1930          return(TIMEOUT);
1931        }
1932
1933        // Mark traversed path segment in global window
1934
1935        global_window->set_color(TRAV_PATH_COLOR);
1936        for (j = 0; j < i - 1; j++) {
1937          global_window->display_line(nav_path.x[ j], nav_path.y[ j],
1938                            nav_path.x[ j + 1], nav_path.y[ j + 1] );
1939        }
1940        global_window->flush();
1941        global_window->set_color("black");
1942
1943        // Mark traversed path segment in robot window
1944
1945        //    for (j = 0; j < i - 1; j++) {
1946        //       draw_line(nav_path.x[ j], nav_path.y[ j],
1947        //          nav_path.x[ j + 1], nav_path.y[ j + 1],
1948        //          ROBOT_TRAV_PATH_COLOR + 2);
1949        //    }
1950      }
1951
1952      st();
1953
1954      return(OK);                              // Arrived at goal
1955    }
1956
1957    int agent::frontier_path_nav_seq(int front_index)      // Frontier index
1958    {
1959      // Navigate to frontier by planning and following path
1960
1961      path nav_path;  // Navigation path
1962      double gx, gy;  // World coords of frontier centroid
1963      int nav_status; // Navigation status
1964      int path_found; // 1 if path found, 0 otherwise
1965      int i, j;
```

```
1966
1967        gx = frontiers[ front_index] .x;
1968        gy = frontiers[ front_index] .y;
1969
1970        path_found = frontier_path_plan(gx, gy, front_index, nav_path);
1971        if (!path_found) {
1972          return(NO_PATH);
1973        }
1974
1975        update();
1976
1977        //  cout << "Press <enter> to continue." << endl;
1978        //  cin.get();
1979
1980        for (i = 1; i < nav_path.length; ) {
1981          nav_status = path_local_nav_seq(nav_path, i);
1982
1983          // Stop immediately at end of path
1984          // (so the robot doesn't crash if the goal is next to a wall)
1985          if (i == nav_path.length) {
1986            st();
1987            cout << "Stopping at path's end." << endl;
1988          }
1989
1990          if (nav_status == ABORT) {              // User aborted
1991            return(ABORT);
1992          }
1993
1994          if (nav_status == ALT) {          // Arrived unexpectedly at goal
1995            display_path(nav_path, TRAV_PATH_COLOR, global_window);
1996            return(OK);
1997          }
1998
1999          if (nav_status == TIMEOUT) {     // Navigation timeout
2000            return(TIMEOUT);
2001          }
2002
2003          // Mark traversed path segment in global window
2004
2005          global_window->set_color(TRAV_PATH_COLOR);
2006          for (j = 0; j < i - 1; j++) {
2007            global_window->display_line(nav_path.x[ j] , nav_path.y[ j] ,
2008                              nav_path.x[ j + 1] , nav_path.y[ j + 1] );
2009          }
2010          global_window->flush();
2011          global_window->set_color("black");
2012
2013          // Mark traversed path segment in robot window
2014
2015          //    for (j = 0; j < i - 1; j++) {
2016          //       draw_line(nav_path.x[ j] , nav_path.y[ j] ,
2017          //              nav_path.x[ j + 1] , nav_path.y[ j + 1] ,
2018          //              ROBOT_TRAV_PATH_COLOR + 2);
2019          //  }
2020        }
2021
2022        return(OK);                              // Arrived at goal
2023      }
```

```
2024
2025    void agent::sonar_sweep_seq(Map3D map)
2026    {
2027       // Rotate sonar sensors and scan
2028
2029       int i;
2030
2031       for (i = 0; i < SONAR_SWEEP_WIDTH; i += SONAR_SWEEP_STEP) {
2032          update();
2033    // THESIS SCOUT CHANGE send r.theta not r.turret for SCOUT
2034          sonar_scan(sonar_smd, sonar_clear_smd, map, r.x, r.y, r.theta);
2035       //    cout << "r.theta= " << r.theta << endl;    // show robot heading
2036    value
2037    //    grid_display_global(map);   // TEMP FIX test map display - shows
2038    updated display after each scan
2039
2040    // BEGIN SCOUT THESIS CHANGE
2041          scout_vm(0, SONAR_SWEEP_STEP * 10);    // Rotate the robot, - not the
2042    turret  - changed pr to vm
2043    //    ws(1, 1, 0, 5);          // TEMP FIX comment this line out and try
2044    sleep instead
2045          sleep(3);    // SCOUT THESIS CHANGE added this line as test **PAUSE
2046    robot at intervals**
2047       }
2048
2049    // SCOUT THESIS CHANGE - do not rotate Scout back as line below would do
2050    //   hopefully this will decrease the odometry error buildup
2051    //   scout_pr(0, SONAR_SWEEP_WIDTH * -10);      // Rotate the robot back
2052    //   ws(1, 1, 0, 5);   // TEMP FIX comment this line out and try sleep
2053    instead
2054    //    sleep(3);    // TEMP FIX added this line as test
2055    // END SCOUT THESIS CHANGE
2056       update();
2057    }
2058
2059    void agent::sonar_sweep_abs_seq(Map3D map)
2060    {
2061       // Rotate sonar sensors and scan (absolute coordinates)
2062
2063       int i;
2064
2065       for (i = 0; i < SONAR_SWEEP_WIDTH; i += SONAR_SWEEP_STEP) {
2066          update();
2067          sonar_scan_abs(sonar_smd, sonar_clear_smd, map, r.x, r.y, r.theta);
2068    //    cout << "r.theta=" << r.theta << endl;    // show robot heading
2069    value
2070    //    grid_display_global(map);    // TEMP FIX test map display - shows
2071    updated display after each scan
2072    // TEMP FIX send r.theta not r.turret for SCOUT
2073
2074    // BEGIN SCOUT THESIS CHANGE
2075          scout_vm(0, SONAR_SWEEP_STEP * 10);        // changed pr to vm
2076    //    ws(1, 1, 0, 5);        // TEMP FIX comment this line out and try
2077    sleep cmd instead
2078          sleep(1);    // TEMP FIX added this line as test
2079       }
2080
```

```
2081   //   SCOUT THESIS CHANGE - do not rotate Scout back as line below would
2082   do
2083   //   hopefully this will decrease the odometry error buildup
2084   //   scout_pr(0, SONAR_SWEEP_WIDTH * -10);
2085   //   ws(1, 1, 0, 5);       // TEMP FIX comment out this line and try sleep
2086   cmd instead
2087   //   sleep(3);    // TEMP FIX added this line as test
2088   // END SCOUT THESIS CHANGE
2089     update();
2090   }
2091
2092   void agent::laser_sweep_seq(Map3D map)
2093   //   Rotate laser scanner and scan
2094   {
2095     int scans = 0;
2096
2097   // BEGIN SCOUT THESIS CHANGE
2098     scout_vm(0, 3600);    // just in case we ever put a laser on the Scout
2099   // TEMP FIX - use vm instead of pr
2100   // END SCOUT THESIS CHANGE
2101     r.wait_start();
2102
2103     while(State[ STATE_VEL_TURRET] > 0) {
2104       laser_scan(map, r.x, r.y, r.theta);    // TEMP FIX for SCOUT if it
2105   ever has a fixed laser -yeh right
2106       if (realtime_display) {
2107         display_robot(global_window, State[ 34] , State[ 35] , State[ 36] ,
2108   State[ 37] );
2109       }
2110       scans++;
2111     }
2112
2113     cout << scans << " scans completed : avg scan interval = "
2114         << 360.0 / (double) scans << " degrees" << endl;
2115   }
2116
2117   void agent::laser_sweep_abs_seq(Map3D map)
2118   //   Rotate laser scanner and scan (absolute coordinates)
2119   {
2120     int scans = 0;
2121
2122   // BEGIN SCOUT THESIS CHANGE
2123     scout_vm(0, 3600);    // TEMP FIX - use vm instead of pr
2124   // END SCOUT THESIS CHANGE
2125     r.wait_start();
2126
2127     while(State[ STATE_VEL_TURRET] > 0) {
2128       laser_scan_abs(map, r.x, r.y, r.theta);   //TEMP FIX for SCOUT with
2129   fixed laser
2130       if (realtime_display) {
2131         display_robot(global_window, State[ 34] , State[ 35] , State[ 36] ,
2132   State[ 37] );
2133       }
2134       scans++;
2135     }
2136
2137     cout << scans << " scans completed : avg scan interval = "
2138         << 360.0 / (double) scans << " degrees" << endl;
```

249

```
2139    }
2140
2141    void agent::lls_sweep_seq(Map3D map)
2142    //  Laser-limited sonar sweep
2143    {
2144       int scans = 0;
2145
2146    // SCOUT NOTE - do not know how Scout handles sp command
2147       sp(DEFAULT_SPEED, DEFAULT_TURN_RATE, 0);   // TEMP FIX for SCOUT
2148
2149       r.sonar_single(0);
2150       r.ir_single(0);
2151    // BEGIN SCOUT THESIS CHANGE
2152       scout_vm(0, 3600);       // TEMP FIX- try vm instead of pr commands for
2153    SCOUT
2154    // END SCOUT THESIS CHANGE
2155       r.wait_start();
2156
2157       while(State[ STATE_VEL_TURRET] > 0) {
2158          lls_scan(sonar_smd, sonar_clear_smd, map, r.x, r.y, r.theta);   //
2159    TEMP FIX for SCOUT
2160          if (realtime_display) {
2161             display_robot(global_window, State[ 34] , State[ 35] , State[ 36] ,
2162    State[ 37] );
2163          }
2164          scans++;
2165       }
2166
2167       cout << scans << " scans completed : avg scan interval = "
2168            << 360.0 / (double) scans << " degrees" << endl;
2169
2170       r.ir_on();
2171       r.sonar_on();
2172       r.set_default_velocity();
2173    }
2174
2175    void agent::lls_sweep_abs_seq(Map3D map)
2176    //  Laser-limited sonar sweep (absolute coordinates)
2177    {
2178       int scans = 0;
2179
2180    // SCOUT NOTE - do not know how Scout would handle sp command
2181       sp(DEFAULT_SPEED, DEFAULT_TURN_RATE, 0);   // TEMP FIX for SCOUT
2182
2183       r.sonar_single(0);
2184       r.ir_single(0);
2185    // BEGIN SCOUT THESIS CHANGE
2186       scout_vm(0, 3600);   // TEMP FIX - try vm instead of pr commands for
2187    SCOUT
2188    // END SCOUT THESIS CHANGE
2189       r.wait_start();
2190
2191       while(State[ STATE_VEL_TURRET] > 0) {
2192          lls_scan_abs(sonar_smd, sonar_clear_smd, map, r.x, r.y, r.theta);
2193    // TEMP FIX for SCOUT
2194          if (realtime_display) {
2195             display_robot(global_window, State[ 34] , State[ 35] , State[ 36] ,
2196    State[ 37] );
```

```
2197          }
2198        scans++;
2199      }
2200
2201      cout << scans << " scans completed : avg scan interval = "
2202           << 360.0 / (double) scans << " degrees" << endl;
2203
2204      r.ir_on();
2205      r.sonar_on();
2206      r.set_default_velocity();
2207    }
2208
2209    void agent::map_seq(void)
2210    {
2211      // Build local grid
2212
2213      char vostr[STRLEN];     // Voice string
2214
2215      st();
2216    // BEGIN SCOUT THESIS CHANGE
2217      ws(1, 1, 0, 5);
2218    // END SCOUT THESIS CHANGE
2219
2220      update();
2221      pnet.place_learn((double) r.x, (double) r.y, (double) r.theta / 10.0);
2222
2223    //   sprintf(vostr, "Building map for place %d.\n", pnet.windex);
2224    //   cout << vostr;
2225    //   tk(vostr);
2226
2227      grid_clear(pnet.unit[pnet.windex].lgrid);
2228
2229      clear_robot(pnet.unit[pnet.windex].lgrid, 0, 0);
2230      sonar_sweep_seq(pnet.unit[pnet.windex].lgrid);
2231    //   laser_sweep_seq(pnet.unit[pnet.windex].lgrid);
2232
2233      grid_display(grid_window, pnet.unit[pnet.windex].lgrid);
2234
2235      cout << "Map complete." << endl;
2236    }
2237
2238    void agent::ident_seq(void)
2239    {
2240        // Place identification sequencer
2241
2242        // Build grid
2243
2244        r.update();
2245        grid_clear(egrid);
2246
2247        clear_robot(egrid, 0, 0);
2248        sonar_sweep_seq(egrid);
2249    //     laser_sweep_seq(egrid);
2250
2251        grid_display(grid_window, egrid);
2252
2253        // Identify grid
2254
```

```
2255        grid_ident_seq();
2256   }
2257
2258   void agent::grid_ident_seq(void)
2259   {
2260     // Grid identification sequencer
2261
2262     char comm_str[ STRLEN] ;              // Communications string
2263     char vostr[ STRLEN] ;        // Voice string
2264     double tx, ty;          // Translation vector
2265     double ttheta;          // Rotation
2266     int ix, iy, itheta;           // Identified position
2267     int ident;                    // Place ident index
2268
2269     ident = pnet.best_match(egrid);
2270
2271     cout << "Untransformed best match = [" << ident << "]" << endl;
2272
2273     ident = pnet.best_trans_match(egrid, tx, ty, ttheta);
2274     cout << endl;
2275
2276     cout << "Transformed best match = [" << ident << "]  (" <<
2277   pnet.unit[ ident] .x
2278        << ", " << pnet.unit[ ident] .y << ")" << endl;
2279     cout << "Transformation = (" << tx << ", " << ty << ") [" << ttheta <<
2280   "]"
2281        << endl;
2282
2283     // Update dead reckoning
2284
2285     gs();
2286     ix = (int) (tx * 120.0 + 0.5);
2287     iy = (int) (ty * 120.0 + 0.5);
2288     itheta = wrap(r.theta + (int) (ttheta * 10.0 + 0.5), 0, 3599);
2289
2290     cout << endl;
2291     cout << "place = " << ident << " : x = " << ix << " : y = " << iy
2292          << " : theta = " << itheta << endl;
2293
2294     sprintf(vostr, "I am at place %d.\n", ident);
2295     tk(vostr);
2296
2297     pnet.windex = ident;
2298     pnet.display();
2299
2300     r.x = ix + (int) (pnet.unit[ ident] .x + 0.5);
2301     r.y = iy + (int) (pnet.unit[ ident] .y + 0.5);
2302     r.theta = itheta;
2303
2304     place_robot(r.x, r.y, r.theta, r.theta);
2305
2306   //  sprintf(comm_str, "%s/grid%d %d %d %d", apndir, ident, ix, iy,
2307   itheta);
2308   //  cout << "comm str = <" << comm_str << ">" << endl;
2309   //  write_comm(COMM_CHANNEL, comm_str, strlen(comm_str) + 1);
2310
2311   //  while(read_comm(COMM_CHANNEL, comm_str, 80) < 1);
2312   //  cout << "reply = < " << comm_str << ">" << endl;
```

252

```
2313    }
2314
2315    int agent::frontier_nav_seq(int front_index)    // Frontier destination
2316    index
2317    {
2318      // Navigate to selected frontier
2319
2320      int nav_status; // Navigation status
2321      char vostr[ STRLEN] ;    // Voice string
2322
2323      cout << "Navigating to frontier [" << front_index << "] -- centroid ("
2324          << (int) frontiers[ front_index] .x << ", "
2325          << (int) frontiers[ front_index] .y << ")" << endl;
2326
2327      sprintf(vostr, "Navigating to frontier %d.\n", front_index);
2328      tk(vostr);
2329
2330      //  grid_display_global(global_grid);
2331      //  grid_display_regions(region_map);
2332      //  display_region_centroids(0.0, 0.0);
2333      //  display_robot_region_centroids();
2334
2335      nav_status = frontier_path_nav_seq(front_index);
2336
2337      if (nav_status == ABORT) {
2338        return(ABORT);
2339      }
2340
2341      if (nav_status == OK) {
2342        sprintf(vostr, "Arrived at frontier %d.\n", front_index);
2343        cout << vostr;
2344        tk(vostr);
2345
2346        if (num_visit == MAX_FRONTIERS) {
2347          cout << "Visited too many frontiers (>" << MAX_FRONTIERS << ")."
2348    << endl;
2349          exit(-1);
2350        }
2351
2352        front_visit[ num_visit] .x = frontiers[ front_index] .x;
2353        front_visit[ num_visit] .y = frontiers[ front_index] .y;
2354        num_visit++;
2355      }
2356
2357      if ((nav_status == TIMEOUT) || (nav_status == NO_PATH)) {
2358        if (num_inac > MAX_FRONTIERS) {
2359          cout << "frontier_nav_seq: Too many inaccessible frontiers (> " <<
2360          MAX_FRONTIERS << ")." << endl;
2361          exit(-1);
2362        }
2363
2364        sleep(1);
2365        sprintf(vostr, "Frontier %d is inaccessible.\n", front_index);
2366        cout << vostr;
2367        tk(vostr);
2368
2369        frontier_copy(front_inac[ num_inac] , frontiers[ front_index] );
2370        num_inac++;
```

```
2371        }
2372
2373        return(nav_status);
2374   }
2375
2376   /********** BEHAVIOR SETS **********/
2377
2378   int agent::reactive_explore_behaviors(void)
2379   {
2380       // Behavior set for reactive exploration
2381
2382       // Returns 1 if new place mapped, 0 otherwise
2383
2384       int net_status = 0;
2385
2386       advance();
2387       avoid();
2388       bump_halt();
2389
2390       net_status = pnet.place_learn((double) r.x, (double) r.y,
2391                                (double) r.theta / 10.0);
2392
2393       if (net_status & NEW_PLACE) {
2394         map_seq();
2395         return(1);
2396       }
2397
2398       return(0);
2399   }
2400
2401   int agent::navigation_behaviors(void)
2402   {
2403       // Behavior set for navigation
2404
2405       int nav_status; // 1 if active path link exists at current location,
2406                       // 0 otherwise
2407
2408       advance();
2409       maintain_heading();
2410       avoid();
2411       bump_halt();
2412
2413       nav_status = follow_path();
2414
2415       pnet.place_recall((double) r.x, (double) r.y, (double) r.theta / 10.0,
2416                    destin);
2417
2418       return(nav_status);
2419   }
2420
2421   int agent::local_navigation_behaviors(int gx, int gy)
2422   {
2423       // Behavior set for local navigation
2424
2425       int nav_status = 0;    // 1: arrived, 0: otherwise
2426
2427       corridor_advance();
2428       return(nav_status);
```

```cpp
2429    }
2430
2431    /********** UTILITY FUNCTIONS **********/
2432
2433    void agent::reset(void)
2434    {
2435       // Reset position and timer
2436
2437       dp(0, 0);
2438       da(0, 0);
2439
2440       timer = 0;
2441    }
2442
2443    void agent::set_defaults(void)
2444    {
2445       // Set default command values
2446
2447       speed_arb->clear();
2448       turn_arb->clear();
2449    }
2450
2451    void agent::update(void)
2452    {
2453       // Update robot state and moving obstacles
2454
2455       int i;
2456
2457       if (timer % 10 == 0) {
2458         cout << "Time = " << timer << endl;
2459         //     power_check();
2460         //     if (logfile != NULL) {
2461         //        *logfile << timer << " " << pnet.num_units << endl;
2462         //     }
2463       }
2464
2465       r.update();
2466
2467       //   for (i = 0; i < NUM_MOB; i++) {
2468       //     mob_list[ i] .update(r.x, r.y);
2469       //   }
2470
2471       //   clear_robot(global_grid, r.x, r.y);
2472
2473       if (realtime_display) {
2474         display_robot(global_window, r.x, r.y, r.theta, r.theta);   // TEMP
2475    FIX for SCOUT
2476       }
2477
2478       //     sonar_print(egrid, 1);
2479
2480       timer++;
2481    }
2482
2483    void agent::execute(void)
2484    {
2485         // Send commands to robot
2486
```

```
2487        int speed_com, turn_com;
2488
2489     //    speed_window.display(speed_arb->votes);
2490     //    turn_window.display(turn_arb->votes);
2491
2492        speed_com = (int) speed_arb->command();
2493        turn_com = (int) (turn_arb->command() * 10.0);
2494
2495        if ((speed_com == 0) && (turn_com == 0)) {
2496           turn_com = (int) (rdrand(-RAND_TURN, RAND_TURN) * 10.0);
2497     //       cout << "Random turn <" << turn_com << ">" << endl;
2498        }
2499
2500        r.move(speed_com, turn_com);
2501
2502        home_dist += (int) speed_arb->command();
2503   }
2504
2505   /********** BEHAVIORS **********/
2506
2507   void agent::bump_halt(void)
2508   {
2509      // Go limp if bumper touched
2510
2511   // BEGIN SCOUT THESIS CHANGE
2512   // comment out the code below that was a hack for a bad bumper
2513   // rearrange code to match original code
2514
2515      char vostr[ STRLEN] ;      // Voice string
2516   //   int touch_offset;      // Rotation offset for touch sensors
2517   //   int abs_touch;         // Absolute index of tripped bumper
2518   //   int sleepflag = 0;     // Do you sleep?
2519      int i;
2520
2521      if (r.touch.max() > 0) {
2522        lp();    // robot motors stop
2523
2524        for (i = 0; i < NUM_TOUCH; i++) {
2525          if (r.touch[ i] ) {
2526            sprintf(vostr, "Contact on bumper %d.", i);
2527            cout << vostr << endl;
2528            tk(vostr);
2529                          }    // close if
2530        }   // close for
2531
2532        sprintf(vostr, "Sleeping for %d seconds.", BUMP_SLEEP);
2533        cout << vostr << endl;
2534        tk(vostr);
2535
2536        sleep(BUMP_SLEEP);
2537      }   // close if
2538   }   // clode bump_halt
2539
2540
2541
2542   // Below was all hack code for the procedure above
2543
2544   // HACK!  On Coyote, ignore multiple bumps on same bumper.
```

```
2545    //
2546    // REMOVE THIS WHEN COYOTE'S BUMPER BOARD IS FIXED
2547    //
2548    //      touch_offset = wrap((int) ((double) (r.theta + r.bumper_offset)
2549    //                          / (double) BUMPER_SEP + 0.5),
2550    //                    NUM_TOUCH);
2551    //      abs_touch = wrap(i + touch_offset, NUM_TOUCH);
2552    //
2553    //      if ((r.id == 1) || !bumped[ abs_touch] ) {
2554    //        lp();
2555    //            sprintf(vostr, "Contact on bumper %d.", abs_touch);
2556    //        cout << vostr << endl;
2557    //        tk(vostr);
2558    //        bumped[ abs_touch] = 1;
2559    //        sleepflag = 1;
2560    //      }
2561    //        }
2562    //      }
2563    //
2564    //
2565    //      if (sleepflag) {
2566    //        sprintf(vostr, "Sleeping for %d seconds.", BUMP_SLEEP);
2567    //        cout << vostr << endl;
2568    //        tk(vostr);
2569    //
2570    //        sleep(BUMP_SLEEP);
2571    //      }
2572    //    }
2573    // }
2574
2575    // END SCOUT THESIS CHANGE
2576
2577
2578    // BEGIN SCOUT THESIS CHANGE
2579    // NOTE - the following procedures recoil and bump_recoil were written
2580    // for the Nomad 200 but are NOT implemented in the code
2581    // The major problem with them on the Nomad 200 is misalignment
2582    // of the turret and the base.
2583    //   They should actually be easier to implement for the Nomad Scout
2584    //   because of it lack of a turret bumpers will always be fixed in
2585    relation
2586    // to the robot.
2587    //   Using these would be better than just using the bump_halt routine
2588    // above which only stops the robot but does not get it away from the
2589    obstacle
2590
2591    void agent::recoil(void)
2592    {
2593        // If touched in forward half, move backward and turn
2594
2595        double spd;
2596        double trn;
2597
2598        if (r.touch.max(15, 5) > 0) {
2599            spd = rdrand(-RECOIL_SPEED, 0.0);
2600            trn = rdrand(-RECOIL_TURN, RECOIL_TURN);
2601
2602            speed_arb->vote(spd, RECOIL_SPEED_SIGMA, RECOIL_WT);
```

```
2603            turn_arb->vote(trn, RECOIL_TURN_SIGMA, RECOIL_WT);
2604
2605            cout << "Recoiling back... (speed = " << spd << ", turn = " << trn
2606                << ")" << endl;
2607        }
2608    }
2609
2610    void agent::bump_recoil(void)
2611    {
2612        // If bumper contact, recoil away
2613
2614        char vostr[ STRLEN] ;    // Voice string
2615        double rel_angle;        // Relative angle from robot to bumper contact
2616        double touch_angle;      // Absolute angle from robot to bumper contact
2617        double tx, ty;   // Coords of bumper contact
2618        int contact_flag = 0; // Contact indicator (0 = no, 1 = yes)
2619        int i;
2620
2621        for (i = 0; i < NUM_TOUCH; i++) {
2622            if (r.touch[ i] ) {
2623                lp();          // Go limp
2624
2625                sprintf(vostr, "Contact on bumper %d.", i);
2626                tk(vostr);
2627
2628                // Compute contact angle
2629
2630    // NOTE - the BUMPER_SEP number here would have to be changed
2631    //              to accomodate the different bumper pattern of the Scout
2632    //              which is not evenly spaced around the robot
2633                rel_angle = (double) (i * BUMPER_SEP) / 10.0;
2634
2635                sprintf(vostr, "Relative angle %.0f.", rel_angle);
2636                cout << vostr << endl;
2637                tk(vostr);
2638
2639                if ((rel_angle <= 90.0) || rel_angle >= 270.0) {
2640                // Recoil back if contact is in forward half of robot
2641
2642                cout << "<<< RECOILING BACK" << endl;
2643                tk("Recoiling back.");
2644
2645    // BEGIN SCOUT THESIS CHANGE
2646                scout_vm(-BUMP_RECOIL, 0);  // TEMP FIX - change pr to vm
2647    //      ws(1, 1, 0, 10);   TEMP FIX - comment out the wait
2648    // END SCOUT THESIS CHANGE
2649                }
2650                else {
2651                // Recoil forward if contact is in rear half of robot
2652
2653                cout << "RECOILING FORWARD >>>" << endl;
2654                tk("Recoiling forward.");
2655
2656    // BEGIN SCOUT THESIS CHANGE
2657                scout_vm(BUMP_RECOIL, 0);  // TEMP FIX - change pr to vm
2658    //      ws(1, 1, 0, 10);   TEMP FIX - comment out the wait
2659    // END SCOUT THESIS CHANGE
2660                }
```

258

```
2661
2662        contact_flag = 1;
2663
2664        break;                // Only recoil from one contact
2665      }
2666    }
2667
2668    // Update global grid for all contacts
2669
2670    if (contact_flag) {
2671      for (i = 0; i < NUM_TOUCH; i++) {
2672        if (r.touch[ i] ) {
2673        // Compute contact position
2674
2675  // NOTE - the BUMPER_SEP number here would have to be changed
2676  //               to accomodate the different bumper pattern of the Scout
2677  //               which is not evenly spaced around the robot
2678        rel_angle = (double) (i * BUMPER_SEP) / 10.0;
2679        touch_angle = angle_wrap((double) r.theta / 10.0 + rel_angle);
2680        tx = (double) r.x + ROBOT_RADIUS * 120.0 * cos(touch_angle *
2681  DEG2RAD);
2682        ty = (double) r.y + ROBOT_RADIUS * 120.0 * sin(touch_angle *
2683  DEG2RAD);
2684
2685        // Update global grid
2686
2687        set_location(global_grid, tx / 120.0, ty / 120.0, SONAR_HEIGHT,
2688  POS);
2689        }
2690      }
2691
2692      grid_display_global(global_grid);
2693    }
2694  }
2695
2696  void agent::maintain_alignment(void)
2697  {
2698    // Realign turret if it is not aligned with base
2699
2700    double dev;             // Deviation between base and turret angles
2701    int align_turn; // Turn required to realign turret
2702
2703  // BEGIN SCOUT THESIS CHANGE
2704  // fake code into thinking nonexistent turret is aligned with base
2705    dev = 0.0;   // fix for SCOUT
2706  //  dev = angle_diff((double) r.theta / 10.0, (double) r.turret / 10.0);
2707
2708    if (dev > MAX_BASE_TURRET_DEV) {
2709      tk("Realigning.");
2710      st();
2711
2712      do {
2713        cout << "REALIGNING:  base = " << r.theta << " : turret = "
2714          << r.turret << " : deviation = " << dev << endl;
2715
2716        align_turn =
2717        (int) (angle_sgn_diff((double) r.theta / 10.0,
2718                        (double) r.turret / 10.0)
```

```
2719                      * 10.0 + 0.5);
2720              cout << "Realignment turn = <" << align_turn << ">" << endl;
2721
2722
2723     // NOTE - no turret on Scout to align, next two lines are ignored on
2724     Scout
2725              scout_vm(0, 0);   // TEMP FIX for SCOUT
2726              ws(0, 0, 1, 100);
2727
2728              update();
2729              dev = 0.0;    // fix for SCOUT
2730     //        dev = angle_diff((double) r.turret / 10.0, (double) r.theta /
2731     10.0);
2732          }
2733          while (dev > MAX_BASE_TURRET_DEV);
2734
2735          cout << "Realignment complete: base = " << r.theta << " : turret = "
2736              << r.turret << " : deviation = " << dev << endl;
2737     // END SCOUT THESIS CHANGE
2738        }
2739     }
2740
2741     int agent::advance(void)
2742     {
2743          // Move forward unless front is blocked (return 1 if blocked, 0
2744     otherwise)
2745
2746          int fwd_min;          // Minimum forward distance
2747          int per_min;          // Minimum peripheral distance
2748          double spd;                 // Desired speed
2749
2750          fwd_min = r.arc[ FWD] ;
2751          per_min = r.range.min(FWD_RT, FWD_LF);
2752
2753          if ((fwd_min <= ADV_STOP_DIST) || (per_min <= ADV_PER_STOP_DIST)) {
2754            speed_arb->vote(0.0, ADV_SPEED_SIGMA, ADV_SPEED_WT);
2755            return(1);
2756          }
2757
2758          if ((fwd_min > ADV_SLOW_DIST) && (per_min > ADV_PER_SLOW_DIST)) {
2759            speed_arb->vote(ADV_SPEED, ADV_SPEED_SIGMA, ADV_SPEED_WT);
2760            return(0);
2761          }
2762
2763          spd = ADV_SPEED;
2764
2765          if (fwd_min <= ADV_SLOW_DIST) {
2766            spd = ADV_SPEED * (double) (fwd_min - ADV_STOP_DIST) /
2767            (double) (ADV_SLOW_DIST - ADV_STOP_DIST);
2768          }
2769
2770          if ((per_min <= ADV_PER_SLOW_DIST) && (spd > ADV_PER_SPEED)) {
2771            spd = ADV_PER_SPEED;
2772          }
2773
2774          speed_arb->vote(spd, ADV_SPEED_SIGMA, ADV_SPEED_WT);
2775          return(0);
2776     }
```

```
2777
2778    int agent::advance_slow(void)
2779    {
2780        // Move forward slowly unless front is blocked
2781        // (return 1 if blocked, 0 otherwise)
2782
2783        int fwd_min;          // Minimum forward distance
2784
2785        fwd_min = r.arc[ FWD] ;
2786
2787        if (fwd_min > ADV_SLOW_STOP_DIST) {
2788            speed_arb->vote(ADV_SLOW_SPEED, ADV_SLOW_SPEED_SIGMA,
2789                        ADV_SLOW_SPEED_WT);
2790            return(0);
2791        }
2792        else {
2793            speed_arb->vote(0.0, ADV_SLOW_SPEED_SIGMA, ADV_SLOW_SPEED_WT);
2794        }
2795    }
2796
2797    void agent::maintain_heading(void)
2798    {
2799        // Maintain current heading
2800
2801        turn_arb->vote(0.0, MH_TURN_SIGMA, MH_TURN_WT);
2802    }
2803
2804    void agent::avoid(void)
2805    {
2806        // Avoid nearby obstacles
2807
2808        int i;
2809        double wt;             // Voting weight for avoidance
2810        double theta; // Obstacle direction
2811
2812        for (i = 0; i < NUM_RANGE; i++) {
2813            if (r.range[ i] < AVOID_DIST) {
2814                wt = AVOID_WT_FACTOR *
2815                    (double) (AVOID_DIST - r.range[ i] ) / (double) AVOID_DIST;
2816                theta = r.sensor2theta(i);
2817                if (theta > 180.0) {
2818                    theta -= 360.0;
2819                }
2820                turn_arb->vote(theta, AVOID_TURN_SIGMA, -wt);
2821            }
2822        }
2823    }
2824
2825    void agent::avoid_bias_left(void)
2826    {
2827        // If front is completely blocked, bias avoidance toward the left
2828    side
2829
2830        if (r.range.max(FWD_RT, FWD_LF) > AVOID_BIAS_DIST) {
2831            return;
2832        }
2833
2834        turn_arb->vote(AVOID_BIAS_ANGLE, AVOID_BIAS_SIGMA, AVOID_BIAS_WT);
```

```
2835    }
2836
2837    void agent::avoid_bias_right(void)
2838    {
2839        // If front is completely blocked, bias avoidance toward the right
2840    side
2841
2842        if (r.range.max(FWD_RT, FWD_LF) > AVOID_BIAS_DIST) {
2843            return;
2844        }
2845
2846        turn_arb->vote(-AVOID_BIAS_ANGLE, AVOID_BIAS_SIGMA, AVOID_BIAS_WT);
2847    }
2848
2849    void agent::follow_wall_right(void)
2850    {
2851        // Align with right wall
2852
2853        double fturn; // Follow turn
2854
2855        if ((r.range.min(BBR, FFR) > FOLLOW_MAX_ALIGN_DIST) ||
2856          (r.arc[ FWD] <= FOLLOW_STOP_DIST)) {
2857            return;
2858        }
2859
2860    //  cout << "min(RT,FWD) = <" << r.range.max(RT,FWD) << ">";
2861
2862        if ((r.arc[ BACK_RT] != r.arc[ FWD_RT]) && (r.arc[ FWD] >
2863    FOLLOW_ABORT)) {
2864            fturn = FOLLOW_TURN_FACTOR * (double) (r.arc[ BACK_RT] -
2865    r.arc[ FWD_RT]);
2866            turn_arb->vote(fturn, FOLLOW_TURN_SIGMA, FOLLOW_WT);
2867        }
2868    //  cout << "" << endl;
2869    }
2870
2871    void agent::follow_wall_left(void)
2872    {
2873        // Align with right wall
2874
2875        double fturn; // Follow turn
2876
2877        if ((r.range.min(BBL, FFL) > FOLLOW_MAX_ALIGN_DIST) ||
2878          (r.arc[ FWD] <= FOLLOW_STOP_DIST)) {
2879            return;
2880        }
2881
2882    //  cout << "min(LF,FWD) = <" << r.range.max(LF,FWD) << ">";
2883
2884        if ((r.arc[ BACK_LF] != r.arc[ FWD_LF]) && (r.arc[ FWD] >
2885    FOLLOW_ABORT)) {
2886            fturn= -FOLLOW_TURN_FACTOR * (double) (r.arc[ BACK_LF] -
2887    r.arc[ FWD_LF]);
2888            turn_arb->vote(fturn, FOLLOW_TURN_SIGMA, FOLLOW_WT);
2889        }
2890    //  cout << "" << endl;
2891    }
2892
```

```
2893    void agent::maintain_distance_right(void)
2894    {
2895        // Maintain desired distance from right wall
2896
2897        int right_min;       // Minimum right range reading
2898        double mdturn;       // Maintain distance turn
2899
2900        if ((r.range.min(BBR, FFR) > FOLLOW_MAX_ALIGN_DIST) ||
2901           (r.arc[ FWD] <= FOLLOW_STOP_DIST)) {
2902           return;
2903        }
2904
2905        right_min = r.range.min(BACK, FWD);
2906
2907        if (right_min != DESIRED_DIST) {
2908           mdturn = MD_TURN_FACTOR * (double) (DESIRED_DIST - right_min);
2909           turn_arb->vote(mdturn, MD_TURN_SIGMA, MD_WT);
2910    //     cout << "right_min = <" << right_min << "> : turning <" <<
2911    cmd[ TURN]
2912    //        << ">" << endl;
2913        }
2914    }
2915
2916    void agent::maintain_distance_left(void)
2917    {
2918        // Maintain desired distance from left wall
2919
2920        int left_min;
2921        double mdturn;         // Maintain distance turn
2922
2923        if ((r.range.min(BBL, FFL) > FOLLOW_MAX_ALIGN_DIST) ||
2924           (r.arc[ FWD] <= FOLLOW_STOP_DIST)) {
2925           return;
2926        }
2927
2928        left_min = r.range.min(FWD, BACK);
2929
2930        if (left_min != DESIRED_DIST) {
2931           mdturn = -MD_TURN_FACTOR * (double) (DESIRED_DIST - left_min);
2932           turn_arb->vote(mdturn, MD_TURN_SIGMA, MD_WT);
2933    //     cout << "left_min = <" << left_min << "> : turning <" << cmd[ TURN]
2934    //        << ">" << endl;
2935        }
2936    }
2937
2938    /********** NAVIGATION BEHAVIORS **********/
2939
2940    int agent::follow_path(void)
2941    {
2942        // Turn to follow path
2943
2944        // Returns 1 if outgoing place link is active, 0 otherwise
2945
2946        double path_angle;         // Angle for navigation
2947
2948        if (pnet.output_valid == 0) {
2949    //     cout << "I'm lost..." << endl;
2950           return(0);
```

```
2951          }
2952
2953          path_angle = angle_sgn_diff(pnet.output, (double) r.theta / 10.0);
2954          turn_arb->vote(path_angle, NAV_SIGMA, NAV_WT * pnet.conf);
2955
2956          return(1);
2957     }
2958
2959     int agent::detect_dest(int destin)
2960     {
2961          // Detect arrival at destination
2962
2963          if (pnet.windex == destin) {
2964             cout << "Arrived at destination." << endl;
2965             tk("Arrived at destination.");
2966             return(1);
2967          }
2968          else {
2969             return(0);
2970          }
2971     }
2972
2973     void agent::goal_orient(int gx, int gy)
2974     {
2975          // Turn toward goal (turn in place if deviation is too high)
2976
2977          double bearing;            // Bearing from robot to goal
2978          double goal_angle;         // Angle between heading and bearing
2979
2980          bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;
2981
2982          //    cout << "goal = (" << gx << ", " << gy << ") : current = (" <<
2983     r.x << ", "
2984          //         << r.y << ") : distance = "
2985          //         << hypot((double) (gy - r.y), (double) (gx - r.x)) / 10.0
2986          //            << " : bearing = " << bearing << endl;
2987
2988          goal_angle = angle_sgn_diff(bearing, (double) r.theta / 10.0);
2989          turn_arb->vote(goal_angle, GOAL_SIGMA, GOAL_WT);
2990
2991          //    cout << "heading = " << (double) r.theta / 10.0 << " :
2992     goal_angle = "
2993          //         << goal_angle << endl;
2994     }
2995
2996     /********** FILE ACCESS FUNCTIONS **********/
2997
2998     void agent::save_net(void)
2999     {
3000        // Save net in directory
3001
3002        char dirname[ STRLEN] ;
3003
3004        cout << "Enter directory name ==> ";
3005        cin >> dirname;
3006
3007        pnet.save_all(dirname);
3008     }
```

```
3009
3010   void agent::load_net(void)
3011   {
3012       // Load net from directory
3013
3014       cout << "Enter directory name ==> ";
3015       cin >> apndir;
3016
3017       pnet.load_all(apndir);
3018       pnet.display();
3019   }
3020
3021   /********** LOCALIZATION FUNCTIONS **********/
3022
3023   double agent::compute_range_err(int image[ NUM_RANGE] , vector rinput)
3024   {
3025       // Compute difference between image and range input
3026
3027       double match_err;
3028       int err_sum = 0;
3029       int i;
3030
3031   //    cout << "image/input:error = ";
3032
3033       for (i = 0; i < NUM_RANGE; i++) {
3034         err_sum += abs(image[ i] - rinput[ i] );
3035   //    cout << image[ i] << "/" << rinput[ i] << ":" <<
3036   //         abs(image[ i] - rinput[ i] ) << " ";
3037       }
3038   //    cout << endl;
3039
3040       match_err = (double) err_sum / (double) (NUM_RANGE * MAX_RANGE);
3041       cout << "match error = " << match_err << endl;
3042
3043       return(match_err);
3044   }
3045
3046   /******************** EVIDENCE GRID DISPLAY FUNCTIONS
3047   ********************/
3048
3049   void agent::grid_display(window *win,      // Window pointer
3050                     Map3D map) // Evidence grid
3051   {
3052     // Display evidence grid in X window
3053
3054     double xd, yd;                   // Display coords
3055     double xscale, yscale, zscale;    // Cell dimensions (tenths of
3056   inches)
3057     int x, y, z;                     // Cell index
3058     int xsize, ysize, zsize;         // Grid dimensions (# cells)
3059     int p;                           // Occupancy probability
3060
3061     win->clear_window();
3062
3063     xsize = map.msize[ 0] ;
3064     ysize = map.msize[ 1] ;
3065     zsize = map.msize[ 2] ;
3066
```

```
3067        xscale = (map.himv[ 0] - map.lomv[ 0] ) * 120.0 / (double) xsize;
3068        yscale = (map.himv[ 1] - map.lomv[ 1] ) * 120.0 / (double) ysize;
3069        zscale = (map.himv[ 2] - map.lomv[ 2] ) * 120.0 / (double) zsize;
3070
3071        //  cout << "Displaying grid (" << xsize << " x " << ysize << " x " <<
3072        zsize
3073        //     << ") : scale = (" << xscale << ", " << yscale << ", " << zscale
3074        << ")"
3075        //       << endl;
3076
3077        z = (int) ((SONAR_HEIGHT + HEIGHT_OFFSET - map.lomv[ 2] ) /
3078                   (map.himv[ 2] - map.lomv[ 2] ) * zsize);
3079
3080        for (y = 0; y < ysize; y++) {
3081          for (x = 0; x < xsize; x++) {
3082            p = map.mapm[ z * xsize * ysize + y * xsize + x] ;
3083
3084            xd = ((double) (x + 0.5) * xscale + map.lomv[ 0] * 120.0);
3085            yd = ((double) (y + 0.5) * yscale + map.lomv[ 1] * 120.0);
3086
3087            if (p > 0) {
3088              win->display_circle(xd, yd, xscale / 4.0);
3089            }
3090            else if (p == 0) {
3091              win->display_point(xd, yd);
3092            }
3093
3094            /*      if (p >= GRID_POS_THRESH) {
3095              win->display_circle(xd, yd, xscale / 4.0);
3096            }
3097            else if (p > GRID_NEG_THRESH) {
3098            if (p > 0) {
3099              win->set_color("blue");
3100              win->display_point(xd, yd);
3101              win->set_color("black");
3102            }
3103            else if (p < 0) {
3104              win->set_color("red");
3105              win->display_point(xd, yd);
3106              win->set_color("black");
3107            }
3108            else {
3109              win->display_point(xd, yd);
3110            }
3111            }*/

3112
3113          }
3114        }
3115
3116        win->draw_arc_buffer();
3117        win->flush();
3118      }
3119
3120      void agent::grid_display_global(Map3D map)        // Evidence grid
3121      {
3122        // Display global evidence grid in X window
3123
3124        double xd, yd;                    // Display coords
```

```
3125      double xscale, yscale, zscale;     // Cell dimensions (tenths of
3126    inches)
3127      int x, y, z;                       // Cell index
3128      int xsize, ysize, zsize;           // Grid dimensions (# cells)
3129      int p;                         // Occupancy probability
3130
3131      global_window->clear_window();
3132
3133      xsize = map.msize[ 0] ;
3134      ysize = map.msize[ 1] ;
3135      zsize = map.msize[ 2] ;
3136
3137      xscale = (map.himv[ 0]  - map.lomv[ 0] ) * 120.0 / (double) xsize;
3138      yscale = (map.himv[ 1]  - map.lomv[ 1] ) * 120.0 / (double) ysize;
3139      zscale = (map.himv[ 2]  - map.lomv[ 2] ) * 120.0 / (double) zsize;
3140
3141      cout << "Displaying grid (" << xsize << " x " << ysize << " x " <<
3142    zsize
3143        << ") : scale = (" << xscale << ", " << yscale << ", " << zscale <<
3144    ")"
3145          << endl;
3146            .
3147      z = (int) ((SONAR_HEIGHT + HEIGHT_OFFSET - map.lomv[ 2] ) /
3148              (map.himv[ 2]  - map.lomv[ 2] ) * zsize);
3149
3150      for (y = 0; y < ysize; y++) {
3151        for (x = 0; x < xsize; x++) {
3152          p = map.mapm[ z * xsize * ysize + y * xsize + x] ;
3153
3154          xd = ((double) (x + 0.5) * xscale + map.lomv[ 0]  * 120.0);
3155          yd = ((double) (y + 0.5) * yscale + map.lomv[ 1]  * 120.0);
3156
3157          if (p > 0) {
3158            global_window->display_circle(xd, yd, xscale / 4.0);
3159          }
3160          else if (p == 0) {
3161            global_window->display_point(xd, yd);
3162          }
3163
3164          /*      if (p >= GRID_POS_THRESH) {
3165            global_window->display_circle(xd, yd, xscale / 4.0);
3166          }
3167          else if (p > GRID_NEG_THRESH) {
3168          if (p > 0) {
3169            global_window->set_color("blue");
3170            global_window->display_point(xd, yd);
3171            global_window->set_color("black");
3172          }
3173          else if (p < 0) {
3174            global_window->set_color("red");
3175            global_window->display_point(xd, yd);
3176            global_window->set_color("black");
3177          }
3178          else {
3179            global_window->display_point(xd, yd);
3180          }
3181          }*/
3182
```

```
3183          }
3184       }
3185
3186       global_window->draw_arc_buffer();
3187       global_window->flush();
3188
3189       global_refresh = 1;
3190    }
3191
3192    void agent::display_place_grid(void)
3193    {
3194       // Display local grid for place
3195
3196       int index;              // Place index
3197
3198       if (pnet.num_units == 0) {
3199         cout << "No places in APN." << endl;
3200         return;
3201       }
3202
3203       cout << "Enter place index [ 0.." << pnet.num_units - 1 << "] ==> ";
3204       cin >> index;
3205
3206       if ((index < 0) || (index >= pnet.num_units)) {
3207         cout << "Nonexistent place." << endl;
3208         return;
3209       }
3210
3211       grid_display(grid_window, pnet.unit[ index] .lgrid);
3212    }
3213
3214    void agent::grid_display_edges(int grid[ GLOBAL_X_RES] [ GLOBAL_Y_RES] )
3215                                       // Colored grid
3216
3217    {
3218       // Display edge segments detected in evidence grid
3219
3220       double xd, yd;              // Display coords
3221       double xscale, yscale;         // Cell dimensions (tenths of
3222    inches)
3223       int x, y;                  // Cell index
3224       int xsize, ysize;              // Grid dimensions (# cells)
3225       int p;                     // Occupancy probability
3226
3227       xsize = GLOBAL_X_RES;
3228       ysize = GLOBAL_Y_RES;
3229
3230       xscale = (GLOBAL_X_MAX - GLOBAL_X_MIN) * 120.0 / (double) xsize;
3231       yscale = (GLOBAL_Y_MAX - GLOBAL_Y_MIN) * 120.0 / (double) ysize;
3232
3233    // cout << "Displaying grid (" << xsize << " x " << ysize << " x " <<
3234    zsize
3235    //    << ") : scale = (" << xscale << ", " << yscale << ", " << zscale
3236    << ")"
3237    //       << endl;
3238
3239       global_window->set_color(EDGE_COLOR);
3240
```

```
3241      for (y = 0; y < ysize; y++) {
3242        for (x = 0; x < xsize; x++) {
3243          p = grid[x][y];
3244
3245          xd = (double) (x + 0.5) * xscale + GLOBAL_X_MIN * 120.0;
3246          yd = (double) (y + 0.5) * yscale + GLOBAL_Y_MIN * 120.0;
3247
3248          if (p > 0) {
3249          global_window->display_circle(xd, yd, xscale / 4.0);
3250          }
3251        }
3252      }
3253
3254      global_window->set_color("black");
3255
3256      global_window->draw_arc_buffer();
3257      global_window->flush();
3258  }
3259
3260  void agent::grid_display_regions(int grid[GLOBAL_X_RES][GLOBAL_Y_RES])
3261                          // Colored grid
3262
3263  {
3264      // Display regions detected in evidence grid
3265
3266      double xd, yd;              // Display coords
3267      double xscale, yscale;          // Cell dimensions (tenths of
3268  inches)
3269      int x, y;                  // Cell index
3270      int xsize, ysize;              // Grid dimensions (# cells)
3271      int p;                  // Occupancy probability
3272
3273      xsize = GLOBAL_X_RES;
3274      ysize = GLOBAL_Y_RES;
3275
3276      xscale = (GLOBAL_X_MAX - GLOBAL_X_MIN) * 120.0 / (double) xsize;
3277      yscale = (GLOBAL_Y_MAX - GLOBAL_Y_MIN) * 120.0 / (double) ysize;
3278
3279  //  cout << "Displaying grid (" << xsize << " x " << ysize << " x " <<
3280  zsize
3281  //      << ") : scale = (" << xscale << ", " << yscale << ", " << zscale
3282  << ")"
3283  //        << endl;
3284
3285      for (x = 0; x < xsize; x++) {
3286        for (y = 0; y < ysize; y++) {
3287          p = grid[x][y];
3288
3289          xd = (double) (x + 0.5) * xscale + GLOBAL_X_MIN * 120.0;
3290          yd = (double) (y + 0.5) * yscale + GLOBAL_Y_MIN * 120.0;
3291
3292          if (p > 0) {
3293          global_window->set_color(color_table[(grid[x][y] - 1) %
3294                                  DISPLAY_COLORS]);
3295          //    cout << "display color = "
3296          //          << color_table[(grid[x][y] - 1) % DISPLAY_COLORS] <<
3297  endl;
3298          global_window->display_circle(xd, yd, xscale / 4.0);
```

```
3299            global_window->set_color("black");
3300              }
3301          }
3302        }
3303
3304      global_window->draw_arc_buffer();
3305      global_window->flush();
3306    }
3307
3308    void agent::display_robot(window *win,     // Window
3309                              int x, int y,   // Robot position (1/10 inch)
3310                              int theta,      // Robot heading (1/10 degree)
3311                              int turret)     // Robot turret angle (1/10 degree)
3312    {
3313      // Display robot in window
3314
3315      // Local constants
3316
3317      const double robot_rad = ROBOT_RADIUS * 120.0; // Robot radius (1/10
3318    inch)
3319      const double half_rad = robot_rad * 0.5;        // Half radius (1/10
3320    inch)
3321      const double tendeg = 0.1 * DEG2RAD;            // 1/10 degree in
3322    radians
3323
3324      static double old_fx, old_fy;           // Old robot position
3325      static double old_ftheta;          // Old robot heading
3326      static double old_fturret;         // Old robot turret angle
3327      static double old_bx, old_by;          // Old endpoint of base line
3328      static double old_tx1, old_ty1,    // Old endpoints of turret line
3329        old_tx2, old_ty2;
3330
3331      double fx, fy;          // Robot position (floating point)
3332      double ftheta;          // Robot heading (floating point)
3333      double fturret;         // Robot turret angle (floating point)
3334      double bx, by;          // Endpoint of base line
3335      double tx1, ty1, tx2, ty2;  // Endpoints of turret line
3336
3337      fx = (double) x;
3338      fy = (double) y;
3339      ftheta = (double) theta;
3340      fturret = (double) turret;
3341
3342      bx = fx + cos(ftheta * tendeg) * half_rad;
3343      by = fy + sin(ftheta * tendeg) * half_rad;
3344
3345      tx1 = fx + cos(fturret * tendeg) * half_rad;
3346      ty1 = fy + sin(fturret * tendeg) * half_rad;
3347
3348      tx2 = fx + cos(fturret * tendeg) * robot_rad;
3349      ty2 = fy + sin(fturret * tendeg) * robot_rad;
3350
3351      if (!global_refresh) {
3352        win->display_xor_circle(old_fx, old_fy, robot_rad);
3353        win->display_xor_line(old_fx, old_fy, old_bx, old_by);
3354        win->display_xor_line(old_tx1, old_ty1, old_tx2, old_ty2);
3355      }
3356      global_refresh = 0;
```

```
3357
3358        win->display_xor_circle(fx, fy, robot_rad);
3359        win->display_xor_line(fx, fy, bx, by);
3360        win->display_xor_line(tx1, ty1, tx2, ty2);
3361
3362        win->flush();
3363
3364        old_fx = fx;
3365        old_fy = fy;
3366        old_ftheta = ftheta;
3367        old_fturret = fturret;
3368
3369        old_bx = bx;
3370        old_by = by;
3371
3372        old_tx1 = tx1;
3373        old_ty1 = ty1;
3374        old_tx2 = tx2;
3375        old_ty2 = ty2;
3376    }
3377
3378    /******************** FRONTIER FUNCTIONS ********************/
3379
3380    void agent::frontier_copy(frontier &f1, frontier f2)
3381    {
3382        // Copy frontier <f2> to frontier <f1>
3383
3384        f1.x = f2.x;
3385        f1.y = f2.y;
3386        f1.size = f2.size;
3387        f1.color = f2.color;
3388    }
3389
3390    void agent::find_frontiers(void)
3391    {
3392        // Find frontiers in global grid
3393
3394        find_frontier_edges(&global_grid, &edge_grid, SONAR_HEIGHT);
3395        find_frontier_regions(edge_grid, SONAR_HEIGHT);
3396        //  grid_display_global(global_grid);
3397        grid_display_regions(region_map);
3398        display_region_centroids(0.0, 0.0);
3399        //  display_robot_region_centroids();
3400    }
3401
3402    void agent::find_frontier_edges(Map3D *raw,      // Raw evidence grid
3403    (pointer)
3404                                    Map3D *edge,      // Frontier edge grid
3405    (pointer)
3406                                    double height)    // Z-coord of edge plane
3407    {
3408        // Find frontier edges in <raw> grid and store them in <edge> grid
3409
3410        int xsize, ysize, zsize;    // Grid dimensions (# cells)
3411        int x, y, z;                // Cell index
3412        int p;                      // Occupancy probability
3413        int unk;                    // Unknown neighbor flag (0 = true)
3414
```

```
3415     xsize = raw->msize[ 0] ;
3416     ysize = raw->msize[ 1] ;
3417     zsize = raw->msize[ 2] ;
3418
3419     if ((xsize != edge->msize[ 0] ) || (ysize != edge->msize[ 1] ) ||
3420         (zsize != edge->msize[ 2] )) {
3421       cout << "find_frontier_edges: Grid size mismatch." << endl;
3422       return;
3423     }
3424
3425     z = (int) ((height + HEIGHT_OFFSET - raw->lomv[ 2] ) /
3426             (raw->himv[ 2]  - raw->lomv[ 2] ) * zsize);
3427
3428     for (x = 1; x < xsize - 1; x++) {
3429       for (y = 1; y < ysize - 1; y++) {
3430         edge->mapm[ z * xsize * ysize + y * xsize + x] = 0;
3431
3432         p = raw->mapm[ z * xsize * ysize + y * xsize + x] ;
3433
3434         if (p < 0) {
3435
3436         // unk = 0 if and only if one of cell (x,y)'s neighbors is unknown
3437
3438         unk = raw->mapm[ z * xsize * ysize + y * xsize + x - 1] *
3439           raw->mapm[ z * xsize * ysize + y * xsize + x + 1] *
3440           raw->mapm[ z * xsize * ysize + (y - 1) * xsize + x] *
3441           raw->mapm[ z * xsize * ysize + (y + 1) * xsize + x] ;
3442
3443         if (unk == 0) {
3444           edge->mapm[ z * xsize * ysize + y * xsize + x] = 1;
3445         }
3446         }
3447
3448         /*      if (p <= GRID_NEG_THRESH) {
3449         if (((raw->mapm[ z * xsize * ysize + y * xsize + x - 1]
3450             > GRID_NEG_THRESH) &&
3451           (raw->mapm[ z * xsize * ysize + y * xsize + x - 1]
3452           < GRID_POS_THRESH)) ||
3453           ((raw->mapm[ z * xsize * ysize + y * xsize + x + 1]
3454           > GRID_NEG_THRESH) &&
3455           (raw->mapm[ z * xsize * ysize + y * xsize + x + 1]
3456           < GRID_POS_THRESH)) ||
3457           ((raw->mapm[ z * xsize * ysize + (y - 1) * xsize + x]
3458             > GRID_NEG_THRESH) &&
3459           (raw->mapm[ z * xsize * ysize + (y - 1) * xsize + x]
3460           < GRID_POS_THRESH)) ||
3461           ((raw->mapm[ z * xsize * ysize + (y + 1) * xsize + x]
3462             > GRID_NEG_THRESH) &&
3463           (raw->mapm[ z * xsize * ysize + (y + 1) * xsize + x]
3464           < GRID_POS_THRESH))) {
3465         edge->mapm[ z * xsize * ysize + y * xsize + x] = 1;
3466         }
3467         }*/
3468
3469       }
3470     }
3471   }
3472
```

```
3473    void agent::find_frontier_regions(Map3D edge,        // Frontier edge
3474    grid
3475                                      double height)  // Z-coord of edge plane
3476    {
3477       // Find frontier regions in <edge> grid and add new frontiers
3478
3479       spread_segment(edge, region_map, height);
3480       analyze_regions(region_map);
3481    }
3482
3483    void agent::spread_segment(Map3D grid,         // Uncolored grid
3484                               int color[ GLOBAL_X_RES] [ GLOBAL_Y_RES] ,
3485                                    // Colored grid
3486                               double height) // Z-coord of edge plane
3487    {
3488       // Segment <grid> image into regions in <color> using spreading
3489    activation
3490
3491       int x, y, z;                // Cell index
3492       int nx, ny;                 // Neighboring cell index
3493       int num_colors = 1;         // Number of colors
3494       int xsize, ysize, zsize;    // Grid dimensions (# cells)
3495       int changed;                // Flag indicating whether cell colors
3496    changed
3497
3498       // Find grid dimensions
3499
3500       xsize = grid.msize[ 0] ;
3501       ysize = grid.msize[ 1] ;
3502       zsize = grid.msize[ 2] ;
3503
3504       z = (int) ((height + HEIGHT_OFFSET - grid.lomv[ 2] ) /
3505               (grid.himv[ 2] - grid.lomv[ 2] ) * zsize);
3506
3507       // Set initial colors
3508
3509       for (x = 0; x < xsize; x++) {
3510         for (y = 0; y < ysize; y++) {
3511           if (grid.mapm[ z * xsize * ysize + y * xsize + x] == 0) {
3512           color[ x] [ y] = 0;
3513           }
3514           else {
3515           color[ x] [ y] = num_colors;
3516           num_colors++;
3517           }
3518         }
3519       }
3520
3521       // Use spreading activation to segment regions
3522
3523       do {
3524         changed = 0;
3525         for (x = 0; x < xsize; x++) {
3526           for (y = 0; y < ysize; y++) {
3527           for (nx = x - 1; nx <= x + 1; nx++) {
3528             for (ny = y - 1; ny <= y + 1; ny++) {
3529               if ((nx >= 0) && (nx < GLOBAL_X_RES) &&
3530                   (ny >= 0) && (ny < GLOBAL_Y_RES)) {
```

273

```
3531                    if ((color[ nx][ ny] > 0) && (color[ nx][ ny] < color[ x][ y] )) {
3532                    color[ x][ y] = color[ nx][ ny] ;
3533                    changed = 1;
3534                    }
3535                  }
3536                }
3537              }
3538            }
3539          }
3540        }
3541     while(changed);
3542   }
3543
3544   void agent::print_region_map(int grid[ GLOBAL_X_RES][ GLOBAL_Y_RES] )
3545                              // Colored grid
3546   {
3547     // Print colored grid cell values
3548
3549     char symbol;               // Cell symbol
3550     int x, y;              // Cell index
3551
3552     cout << endl;
3553     for (x = 0; x < GLOBAL_X_RES;x++) {
3554       for (y = 0; y < GLOBAL_Y_RES; y++) {
3555         if (grid[ x][ y] == 0) {
3556         cout << ".";
3557         }
3558         else {
3559         if (grid[ x][ y] < 10) {
3560           symbol = '0' + (char) grid[ x][ y] ;
3561         }
3562         else if (grid[ x][ y] < 36) {
3563           symbol = 'A' + (char) (grid[ x][ y]  - 10);
3564         }
3565         else {
3566           symbol = 'a' + (char) (grid[ x][ y]  - 36);
3567         }
3568         cout << symbol;
3569         }
3570       }
3571       cout << endl;
3572     }
3573     cout << endl;
3574   }
3575
3576   void agent::analyze_regions(int grid[ GLOBAL_X_RES][ GLOBAL_Y_RES] )
3577                              // Colored grid
3578   {
3579     // Determine size and centroid of frontier regions
3580
3581     double xscale, yscale;              // Cell dimensions (tenths of
3582   inches)
3583     double cx, cy;                  // Centroid of new region
3584     int count[ MAX_COLORS] ;              // Edge cell counter for regions
3585     int x_sum[ MAX_COLORS] ;              // Sum of cell x-coords
3586     int y_sum[ MAX_COLORS] ;              // Sum of cell y-coords
3587     int x, y;                  // Cell index
3588     int i;
```

```
3589
3590      num_front = 0;
3591
3592      xscale = (GLOBAL_X_MAX - GLOBAL_X_MIN) * 120.0 / (double)
3593   GLOBAL_X_RES;
3594      yscale = (GLOBAL_Y_MAX - GLOBAL_Y_MIN) * 120.0 / (double)
3595   GLOBAL_Y_RES;
3596
3597      for (i = 0; i < MAX_COLORS; i++) {
3598        count[ i] = 0;
3599        x_sum[ i] = 0;
3600        y_sum[ i] = 0;
3601      }
3602
3603      for (x = 0; x < GLOBAL_X_RES; x++) {
3604        for (y = 0; y < GLOBAL_Y_RES; y++) {
3605          if (grid[ x][ y] > 0) {
3606          count[ grid[ x][ y]]++;
3607          x_sum[ grid[ x][ y]]  += x;
3608          y_sum[ grid[ x][ y]]  += y;
3609          }
3610        }
3611      }
3612
3613      for (i = 1; i < MAX_COLORS; i++) {
3614        if (count[ i] >= MIN_REGION_SIZE) {
3615          cx =
3616          xscale * (double) x_sum[ i] / (double) count[ i] + GLOBAL_X_MIN *
3617   120.0;
3618          cy =
3619          yscale * (double) y_sum[ i] / (double) count[ i] + GLOBAL_Y_MIN *
3620   120.0;
3621
3622          if (!(visited(cx, cy) || inaccessible(cx, cy))) {
3623          //       if (!inaccessible(cx, cy)) {
3624          if (num_front == MAX_FRONTIERS) {
3625            cout << "analyze_regions: Too many regions (>" << MAX_FRONTIERS
3626                  << ")." << endl;
3627          }
3628          else {
3629            frontiers[ num_front] .color = i;
3630            frontiers[ num_front] .size = count[ i] ;
3631            frontiers[ num_front] .x = cx;
3632            frontiers[ num_front] .y = cy;
3633            num_front++;
3634          }
3635          }
3636        }
3637      }
3638
3639      for (i = 0; i < num_front; i++) {
3640        cout << "Region [" << i << "] : size = " << frontiers[ i] .size
3641            << " : centroid = (" << frontiers[ i] .x << ", " << frontiers[ i] .y
3642            << ")" << endl;
3643      }
3644   }
3645
3646   int agent::visited(double cx, double cy)  // Centroid of new region
```

```
3647   {
3648       // Check whether centroid corresponds to previously visited frontier
3649       // Return 1 if visited, 0 otherwise
3650
3651       double dist;          // Distance from new region to visited frontier
3652       int i;
3653
3654       //  cout << "Checking (" << cx << ", " << cy << ") against visited
3655   list."
3656       //          << endl;
3657
3658       for (i = 0; i < num_visit; i++) {
3659           dist = hypot(cx - front_visit[ i] .x, cy - front_visit[ i] .y);
3660           //     cout << "front_visit[" << i << "] at (" << front_visit[ i] .x <<
3661   ", "
3662           //          << front_visit[ i] .y << ") : distance = " << dist;
3663           if (dist <= VISIT_RADIUS) {
3664               //          cout << " [ - VISITED -]" << endl;
3665               return(1);
3666           }
3667           //     cout << endl;
3668       }
3669
3670       return(0);
3671   }
3672
3673   int agent::inaccessible(double cx, double cy)    // Centroid of new
3674   region
3675   {
3676       // Check whether centroid corresponds to inaccessible frontier
3677       // Return 1 if inaccessible, 0 otherwise
3678
3679       double dist;          // Distance from new region to inaccessible
3680   frontier
3681       int i;
3682
3683       //  cout << "Checking (" << cx << ", " << cy << ") against
3684   inaccessible list."
3685       //          << endl;
3686
3687       for (i = 0; i < num_inac; i++) {
3688           dist = hypot(cx - front_inac[ i] .x, cy - front_inac[ i] .y);
3689           //     cout << "front_inac[" << i << "] at (" << front_inac[ i] .x <<
3690   ", "
3691           //          << front_inac[ i] .y << ") : distance = " << dist;
3692           if (dist <= INAC_RADIUS) {
3693               //          cout << " [ * INACCESSIBLE *]" << endl;
3694               return(1);
3695           }
3696           //     cout << endl;
3697       }
3698
3699       return(0);
3700   }
3701
3702   int agent::closest_frontier(double x, double y)
3703   {
3704       // Return index of unvisited, accessible frontier closest to (x, y)
```

```
3705      // Return -1 if no such frontier exists
3706
3707      double min_dist = MAX_DIST; // Minimum distance to frontier
3708      double dist = -1;            // Distance to frontier
3709      int close_index = -1;        // Index of closest frontier
3710      int i;
3711
3712      for (i = 0; i < num_front; i++) {
3713        if (!(visited(frontiers[ i] .x, frontiers[ i] .y) ||
3714            inaccessible(frontiers[ i] .x, frontiers[ i] .y))) {
3715          //    if (!inaccessible(frontiers[ i] .x, frontiers[ i] .y)) {
3716          dist = hypot(x - frontiers[ i] .x, y - frontiers[ i] .y);
3717          if (dist < min_dist) {
3718          min_dist = dist;
3719          close_index = i;
3720          }
3721        }
3722      }
3723
3724      return(close_index);
3725    }
3726
3727    void agent::display_region_centroids(double cx, // Display center x-
3728    coord
3729                                      double cy)    // Display center y-coord
3730    {
3731      // Mark region centroids in evidence grid window
3732
3733      double xd, yd;              // Display coords
3734      char label[ STRLEN] ;              // Mark label (index)
3735      int mark_color;            // Mark color
3736      int i;
3737
3738      for (i = 0; i < num_front; i++) {
3739        xd = frontiers[ i] .x - cx;
3740        yd = frontiers[ i] .y - cy;
3741
3742        mark_color = (frontiers[ i] .color - 1) % DISPLAY_COLORS;
3743        //    cout << "Drawing frontier [" << i << "] in " <<
3744    color_table[ mark_color]
3745        //         << " (" << mark_color << ")" << endl;
3746
3747        global_window->set_color(color_table[ mark_color] );
3748        global_window->display_circle(xd, yd, CENTROID_MARK_RADIUS);
3749        global_window->display_line(xd - CENTROID_MARK_RADIUS, yd,
3750                          xd + CENTROID_MARK_RADIUS, yd);
3751        global_window->display_line(xd, yd - CENTROID_MARK_RADIUS,
3752                          xd, yd + CENTROID_MARK_RADIUS);
3753
3754        sprintf(label, "%d", i);
3755        global_window->display_text(xd + CENTROID_MARK_RADIUS * 2.0, yd,
3756    label);
3757        global_window->set_color("black");
3758      }
3759      global_window->flush();
3760      //   cout << endl;
3761
3762      //   for (i = 0; i < DISPLAY_COLORS; i++) {
```

```
3763      //     global_window->set_color(color_table[ i] );
3764      //     global_window->display_line(0, i * -100, 1000, i * -100);
3765      //  }
3766      //  global_window->set_color("black");
3767      //  global_window->flush();
3768    }
3769
3770    void agent::display_robot_region_centroids(void)
3771    {
3772      // Mark region centroids in robot window
3773
3774      int xd, yd;                        // Display coords
3775      int mark_color;             // Mark color
3776      int color_mode;             // Color mode for draw command
3777      int i;
3778
3779      refresh_all();
3780
3781      for (i = 0; i < num_front; i++) {
3782        xd = (int) frontiers[ i] .x;
3783        yd = (int) frontiers[ i] .y;
3784
3785        //     mark_color = (frontiers[ i] .color - 1) % DISPLAY_COLORS;
3786        //     color_mode = robot_color[ mark_color] + 2;
3787        //     cout << "Drawing frontier [ " << i << "] in " <<
3788    color_table[ mark_color]
3789        //          << " (" << robot_color[ mark_color] << ") [ mode " <<
3790    color_mode << "] "
3791        //          << endl;
3792
3793        color_mode = 1;
3794
3795        draw_arc(xd - (int) CENTROID_MARK_RADIUS, yd + (int)
3796    CENTROID_MARK_RADIUS,
3797               (int) (CENTROID_MARK_RADIUS * 2.0),
3798               (int) (CENTROID_MARK_RADIUS * 2.0),
3799               0, 3600, color_mode);
3800        draw_line(xd - (int) CENTROID_MARK_RADIUS, yd,
3801               xd + (int) CENTROID_MARK_RADIUS, yd, color_mode);
3802        draw_line(xd, yd - (int) CENTROID_MARK_RADIUS,
3803               xd, yd + (int) CENTROID_MARK_RADIUS, color_mode);
3804      }
3805      //  cout << endl;
3806
3807      //  for (i = 0; i < DISPLAY_COLORS; i++) {
3808      //    color_mode = robot_color[ i] + 2;
3809      //    draw_line(0, i * -100, 1000, i * -100, color_mode);
3810      //  }
3811    }
3812
3813    int agent::check_frontier_cell(int x, int y,     // Cell index
3814                            int front_index) // Frontier index
3815    {
3816      // Check whether cell (x, y) is part of frontier <front_index>
3817
3818      if (frontiers[ front_index] .color == region_map[ x] [ y] ) {
3819        return(1);
3820      }
```

278

```
3821      else {
3822        return(0);
3823      }
3824    }
3825
3826    /******************** NAVIGATION FUNCTIONS ********************/
3827
3828    void agent::corridor_advance(void)
3829    {
3830      // Move forward if front corridor is clear
3831
3832      if (wide_corridor[ FWD] == 1) {
3833    // TEMP FIX for SCOUT comment out mv command below and change to
3834    scout_vm
3835    //    mv(MV_VM, CORRIDOR_SPEED_WIDE, MV_IGNORE, 0, MV_IGNORE, 0);
3836    cout << "In corridor_advance wide_corridor about to call scout_vm ("
3837            << CORRIDOR_SPEED_WIDE << ", 0" << endl;  // TEMP FIX for SCOUT
3838        scout_vm(CORRIDOR_SPEED_WIDE, 0);  // TEMP FIX for SCOUT
3839      }
3840      else if (corridor[ FWD] == 1) {
3841    // TEMP FIX for SCOUT comment out mv command below and change to
3842    scout_vm·
3843    //    mv(MV_VM, CORRIDOR_SPEED, MV_IGNORE, 0, MV_IGNORE, 0);
3844    cout << "In corridor_advance corridor about to call scout_vm ("
3845            << CORRIDOR_SPEED << ", 0" << endl;  // TEMP FIX for SCOUT
3846        scout_vm(CORRIDOR_SPEED, 0);  // TEMP FIX for SCOUT
3847      }
3848      else {
3849    // TEMP FIX for SCOUT comment out mv command below and change to
3850    scout_vm
3851    //    mv(MV_VM, 0, MV_IGNORE, 0, MV_IGNORE, 0);
3852    cout << "In corridor_advance else about to call scout_vm (0,0" << endl;
3853    // TEMP FIX for SCOUT
3854        scout_vm(0, 0);  // TEMP FIX for SCOUT
3855      }
3856    }
3857
3858    void agent::goal_corridor_orient(int gx, int gy)
3859    {
3860        // Turn toward clear corridor closest to goal bearing
3861
3862        double bearing;            // Bearing from robot to goal
3863        double corridor_index;     // Index of selected corridor (-1 = none)
3864        double corridor_bearing;   // Bearing of selected corridor
3865        double cmd_bearing;        // Bearing to face
3866
3867        update();
3868
3869        bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;
3870
3871        //    cout << "goal = (" << gx << ", " << gy << ") : current = (" <<
3872    r.x << ", "
3873        //        << r.y << ") : distance = "
3874        //        << hypot((double) (gy - r.y), (double) (gx - r.x)) / 10.0
3875        //          << " : bearing = " << bearing << endl;
3876
3877        detect_corridors();
3878        corridor_index = select_corridor(bearing);
```

```
3879         corridor_bearing =
3880           angle_wrap((double) (corridor_index * SENSOR_SEP + r.theta) /
3881     10.0);
3882
3883         if ((corridor_index == -1) ||
3884            (angle_diff(bearing, corridor_bearing) > CORRIDOR_MAX_DEVIATION))
3885     {
3886           cmd_bearing =
3887           angle_wrap((double) r.theta / 10.0 +
3888                       rdrand(-GOAL_CORRIDOR_NOISE, GOAL_CORRIDOR_NOISE));
3889         }
3890         else {
3891           cmd_bearing =
3892           angle_wrap(corridor_bearing +
3893                       rdrand(-GOAL_CORRIDOR_NOISE, GOAL_CORRIDOR_NOISE));
3894         }
3895
3896         //     cout << "corridor index = " << corridor_index << " : corridor
3897     bearing = "
3898         //          << corridor_bearing << " : command bearing = " <<
3899     cmd_bearing << endl;
3900
3901         r.face_angle_fast((int) (cmd_bearing * 10.0));    // TEMP FIX for
3902     SCOUT
3903     }
3904
3905     void agent::update_nav_grid(void)
3906     {
3907       // Update navigation grid based on global grid
3908
3909       // grid_fine_to_coarse(global_grid, nav_grid);
3910
3911       grid_copy(nav_grid, global_grid);
3912
3913       //  grid_display(nav_window, nav_grid);
3914     }
3915
3916     int agent::path_plan(double wx, double wy,      // World coords of goal
3917                    path &nav_path)           // Navigation path (optimized)
3918     {
3919       // Plan path to goal location (return 1 if path found, 0 otherwise)
3920
3921       path rev_path;   // Reversed path
3922       path unopt_path;       // Unoptimized navigation path
3923       path opt_path;   // Optimized navigation path
3924       int gx, gy, gz; // Grid coordinates of destination
3925       int rx, ry, rz; // Grid coordinates of robot
3926       int x, y;        // Cell index
3927       int search_status;      // Flag indicating whether path has been found
3928
3929       world2grid(nav_grid, (double) r.x / 120.0, (double) r.y / 120.0,
3930                0, &rx, &ry, &rz);
3931
3932       world2grid(nav_grid, wx / 120.0, wy / 120.0, 0, &gx, &gy, &gz);
3933
3934       cout << "Robot location = (" << r.x << ", " << r.y << ") [" << rx <<
3935     ", "
3936               << ry << "]" << endl;
```

```
3937        cout << "Goal location = (" << wx << ", " << wy << ") [" << gx << ", "
3938             << gy << "]" << endl;
3939
3940        update_nav_grid();
3941
3942        for (x = 0; x < NAV_X_RES; x++) {
3943          for (y = 0; y < NAV_Y_RES; y++) {
3944            visit[x][y] = 0;
3945          }
3946        }
3947
3948        search_status = find_path(rx, ry, gx, gy, rev_path);
3949
3950        if (search_status == SEARCH_FAIL) {
3951          cout << "No path found." << endl;
3952          return(0);
3953        }
3954
3955        reverse_path(rev_path, unopt_path);
3956
3957        cout << "Unoptimized: ";
3958        print_path(unopt_path);
3959
3960        optimize_path(unopt_path, opt_path);
3961        cout << "Optimized: ";
3962        print_path(opt_path);
3963
3964        generate_world_path(opt_path, nav_path);
3965        cout << "World path:";
3966        print_path(nav_path);
3967
3968        //  grid_display(nav_window, nav_grid);
3969        //  display_path(nav_path, OPT_PATH_COLOR, nav_window);
3970        display_path(nav_path, OPT_PATH_COLOR, global_window);
3971        //  display_path_robot(nav_path, ROBOT_OPT_PATH_COLOR);
3972
3973        return(1);
3974      }
3975
3976    int agent::frontier_path_plan(double wx, double wy, // World coords of
3977    goal
3978                                  int front_index,      // Frontier index
3979                                  path &nav_path)       // Navigation path
3980    {
3981        // Plan path to goal location (return 1 if path found, 0 otherwise)
3982
3983        path rev_path;  // Reversed path
3984        path unopt_path;        // Unoptimized navigation path
3985        path opt_path;  // Optimized navigation path
3986        int gx, gy, gz; // Grid coordinates of destination
3987        int rx, ry, rz; // Grid coordinates of robot
3988        int x, y;       // Cell index
3989        int search_status;      // Flag indicating whether path has been found
3990
3991        world2grid(nav_grid, (double) r.x / 120.0, (double) r.y / 120.0,
3992                   0, &rx, &ry, &rz);
3993
3994        world2grid(nav_grid, wx / 120.0, wy / 120.0, 0, &gx, &gy, &gz);
```

```
3995
3996      cout << "Robot location = (" << r.x << ", " << r.y << ") [" << rx <<
3997   ", "
3998           << ry << "]" << endl;
3999      cout << "Goal location = (" << wx << ", " << wy << ") [" << gx << ", "
4000           << gy << "]" << endl;
4001
4002      update_nav_grid();
4003
4004      for (x = 0; x < NAV_X_RES; x++) {
4005        for (y = 0; y < NAV_Y_RES; y++) {
4006          visit[x][y] = 0;
4007        }
4008      }
4009
4010      search_status = frontier_find_path(rx, ry, gx, gy, front_index,
4011   rev_path);
4012
4013      if ((search_status == SEARCH_FAIL) || (search_status ==
4014   SEARCH_TIMEOUT)) {
4015        cout << "No path found." << endl;
4016        return(0);
4017      }
4018
4019      reverse_path(rev_path, unopt_path);
4020
4021      cout << "Unoptimized: ";
4022      print_path(unopt_path);
4023
4024      optimize_path(unopt_path, opt_path);
4025      cout << "Optimized: ";
4026      print_path(opt_path);
4027
4028      generate_world_path(opt_path, nav_path);
4029      cout << "World path:";
4030      print_path(nav_path);
4031
4032      //  grid_display(nav_window, nav_grid);
4033      //  display_path(nav_path, OPT_PATH_COLOR, nav_window);
4034      display_path(nav_path, OPT_PATH_COLOR, global_window);
4035      //  display_path_robot(nav_path, ROBOT_OPT_PATH_COLOR);
4036
4037      return(1);
4038   }
4039
4040   void agent::print_path(path p)
4041   {
4042      // Print all cells on path
4043
4044      int i;
4045
4046      cout << "path length = " << p.length << " : path = ";
4047
4048      for (i = 0; i < p.length; i++) {
4049        cout << "(" << p.x[i] << ", " << p.y[i] << ") ";
4050      }
4051
4052      cout << endl;
```

```
4053    }
4054
4055    void agent::display_path(path p,            // Path
4056                            char *pcolor,      // Path color
4057                            window *win)       // Window
4058    {
4059        // Draw path in window
4060
4061        int i;
4062
4063        win->set_color(pcolor);
4064
4065        for (i = 0; i < p.length - 1; i++) {
4066            win->display_line(p.x[i], p.y[i], p.x[i + 1], p.y[i + 1]);
4067        }
4068
4069        win->flush();
4070        win->set_color("black");
4071    }
4072
4073    void agent::display_path_robot(path p,              // Path
4074                                  int pcolor)          // Path color
4075    {
4076        // Draw path in robot window
4077
4078        int i;
4079
4080        for (i = 0; i < p.length - 1; i++) {
4081            draw_line(p.x[i], p.y[i], p.x[i + 1], p.y[i + 1], pcolor + 2);
4082        }
4083    }
4084
4085    int agent::find_path(int sx, int sy,       // Start cell
4086                         int gx, int gy,       // Goal cell
4087                         path &p)              // Path
4088    {
4089        // Find path from (sx, sy) to (gx, gy)
4090
4091        int nx, ny;                // Neighbor cell index
4092
4093        path_init(p);
4094
4095        visit[sx][sy] = 1;
4096
4097        while(closest_neighbor(sx, sy, gx, gy, nx, ny)) {
4098            if (search_cell(nx, ny, gx, gy, p) == SEARCH_SUCCESS) {
4099                //      cout << "[ ON PATH (" << x << ", " << y << ") ]" << endl;
4100                path_add(p, sx, sy);
4101                return(SEARCH_SUCCESS);
4102            }
4103        }
4104
4105        return(SEARCH_FAIL);
4106    }
4107
4108    int agent::frontier_find_path(int sx, int sy,    // Start cell
4109                                  int gx, int gy,    // Goal cell
4110                                  int front_index,   // Frontier index
```

283

```
4111                              path &p)              // Path
4112    {
4113       // Find path from (sx, sy) to (gx, gy) or any point on frontier
4114    <front_index>
4115
4116       int nx, ny;           // Neighbor cell index
4117       int status;           // Cell search status
4118
4119       path_init(p);
4120
4121       visit[ sx][ sy] = 1;
4122
4123       while(closest_neighbor(sx, sy, gx, gy, nx, ny)) {
4124          cell_count = 0;
4125          status = frontier_search_cell(nx, ny, gx, gy, front_index, p);
4126          if (status == SEARCH_SUCCESS) {
4127             //      cout << "[ ON PATH (" << x << ", " << y << ") ]" << endl;
4128             path_add(p, sx, sy);
4129             return(SEARCH_SUCCESS);
4130          }
4131          if (status == SEARCH_TIMEOUT) {
4132             return(SEARCH_TIMEOUT);
4133          }
4134       }
4135
4136       return(SEARCH_FAIL);
4137    }
4138
4139    int agent::search_cell(int x, int y,      // Search cell
4140                           int gx, int gy,   // Goal cell
4141                           path &p)          // Path
4142    {
4143       // Search cell (x,y) and return search status
4144
4145       int status;                     // Search status
4146       int nx, ny;                     // Neighbor cell index
4147
4148       if (visit[ x][ y] ) {
4149          cout << "search_cell: Error: revisited cell (" << x << ", " << y <<
4150    ")"
4151             << endl;
4152          exit(-1);
4153       }
4154       visit[ x][ y] = 1;
4155
4156       //  cout << "Searching (" << x << ", " << y << ") : ";
4157
4158       if ((x < 0) || (x >= NAV_X_RES) || (y < 0) || (y >= NAV_Y_RES)) {
4159          //    cout << "Out of bounds." << endl;
4160          return(SEARCH_FAIL);
4161       }
4162
4163
4164       if ((x == gx) && (y == gy)) {
4165          //    cout << "[ * GOAL (" << x << ", " << y << ") *]" << endl;
4166          path_add(p, x, y);
4167          return(SEARCH_SUCCESS);
4168       }
```

```
4169
4170     if ((nav_grid.mapm[ grid2index(nav_grid, x, y, 0)] >= 0) ||
4171         (!check_clear(x, y))) {
4172       //     cout << "> BLOCKED <" << endl;
4173       return(SEARCH_FAIL);
4174     }
4175
4176     //  cout << "(( Searching adjacent ))" << endl;
4177
4178     while(closest_neighbor(x, y, gx, gy, nx, ny)) {
4179       if (search_cell(nx, ny, gx, gy, p) == SEARCH_SUCCESS) {
4180         //       cout << "[ ON PATH (" << x << ", " << y << ") ]" << endl;
4181         path_add(p, x, y);
4182         return(SEARCH_SUCCESS);
4183       }
4184     }
4185
4186     return(SEARCH_FAIL);
4187 }
4188
4189 int agent::frontier_search_cell(int x, int y,          // Search cell
4190                                 int gx, int gy,        // Goal cell
4191                                 int front_index,   // Frontier index
4192                                 path &p)           // Path
4193 {
4194     // Search cell (x,y) while navigating to frontier and return search
4195 status
4196
4197     int child_status;          // Search status for child cell
4198     int nx, ny;                // Neighbor cell index
4199
4200     cell_count++;
4201     if (cell_count % 100 == 0) {
4202       cout << "Searching " << cell_count << " cells..." << endl;
4203     }
4204     if (cell_count > SEARCH_MAX_CELLS) {
4205       cell_count = 0;
4206       return(SEARCH_TIMEOUT);
4207     }
4208
4209     if (visit[ x][ y] ) {
4210       cout << "frontier_search_cell: Error: revisited cell (" << x << ", "
4211 << y << ")"
4212           << endl;
4213       exit(-1);
4214     }
4215     visit[ x][ y]  = 1;
4216
4217     //  cout << "Searching (" << x << ", " << y << ") : ";
4218
4219     if ((x < 0) || (x >= NAV_X_RES) || (y < 0) || (y >= NAV_Y_RES)) {
4220       //     cout << "Out of bounds." << endl;
4221
4222       return(SEARCH_FAIL);
4223     }
4224
4225
4226     //  if ((x == gx) && (y == gy)) {
```

```
4227
4228    if (((x == gx) && (y == gy)) ||
4229         (check_frontier_arrival(x, y, front_index))) {
4230    //     cout << "[ * GOAL (" << x << ", " << y << ") *]" << endl;
4231      path_add(p, x, y);
4232      return(SEARCH_SUCCESS);
4233    }
4234
4235    if ((nav_grid.mapm[ grid2index(nav_grid, x, y, 0)] >= 0) ||
4236         (!check_clear(x, y))) {
4237    //     cout << "> BLOCKED <" << endl;
4238      return(SEARCH_FAIL);
4239    }
4240
4241    //  cout << "(( Searching adjacent ))" << endl;
4242
4243    while(closest_neighbor(x, y, gx, gy, nx, ny)) {
4244      child_status = frontier_search_cell(nx, ny, gx, gy, front_index, p);
4245      if (child_status == SEARCH_SUCCESS) {
4246    //        cout << "[ ON PATH (" << x << ", " << y << ") ]" << endl;
4247        path_add(p, x, y);
4248        return(SEARCH_SUCCESS);
4249      }
4250      if (child_status == SEARCH_TIMEOUT) {
4251       return(SEARCH_TIMEOUT);
4252      }
4253    }
4254
4255    return(SEARCH_FAIL);
4256 }
4257
4258 int agent::closest_neighbor(int x, int y,        // Current cell index
4259                             int gx, int gy,       // Goal cell index
4260                             int &nx, int &ny)     // Next cell index
4261 {
4262    // Find index of (unvisited) neighbor closest to goal
4263
4264    double min_dist;        // Minimum distance from neighbor to goal
4265    double dist;            // Distance from neighbor to goal
4266    int found = 0;   // 1 if unvisited neighbor exists, 0 otherwise
4267    int dx, dy;             // Adjacent cell offset
4268    int ax, ay;             // Adjacent cell index
4269
4270    min_dist = (double) MAX_PATH_LENGTH;
4271
4272    for (dx = -1; dx <= 1; dx++) {
4273      for (dy = -1; dy <= 1; dy++) {
4274        ax = x + dx;
4275        ay = y + dy;
4276
4277        if ((ax >= 0) && (ax < NAV_X_RES) && (ay >= 0) && (ay <
4278 NAV_Y_RES)) {
4279          if (visit[ ax][ ay] == 0) {
4280            dist = hypot(gx - ax, gy - ay);
4281            if (dist < min_dist) {
4282              min_dist = dist;
4283              nx = ax;
4284              ny = ay;
```

```
4285                    found = 1;
4286                  }
4287              }
4288            }
4289          }
4290        }
4291
4292      return(found);
4293  }
4294
4295  void agent::reverse_path(path old_path,          // Initial path
4296                     path &new_path)   // Reversed path
4297  {
4298      // Reverse order of steps on path
4299
4300      int i;
4301
4302      path_init(new_path);
4303      for (i = 0; i < old_path.length; i++) {
4304        path_add(new_path, old_path.x[ (old_path.length - 1) - i],
4305              old_path.y[ (old_path.length - 1) - i] );
4306      }
4307  }
4308
4309  void agent::optimize_path(path old_path,   // Initial path
4310                     path &new_path)  // Optimized path
4311  {
4312      // Optimize path by jumping between adjacent path cells
4313
4314      int marker = 0; // Point along old path
4315      int jump_marker;       // Point to jump to on new path
4316      int jump_flag;   // Set to 1 if path jumps
4317
4318      path_init(new_path);
4319      path_add(new_path, old_path.x[ 0] , old_path.y[ 0] );
4320
4321      //  cout << "Starting at (" << new_path.x[ 0] << ", " << new_path.y[ 0]
4322      //       << ") [ 0] <0>" << endl;
4323
4324      while(marker < old_path.length - 1) {
4325        jump_flag = 0;
4326        //     cout << "Trying to jump from (" << new_path.x[ new_path.length
4327  - 1] << ", "
4328        //          << new_path.y[ new_path.length - 1] << ") [" <<
4329  new_path.length - 1
4330        //          << "] <" << marker << ">" << endl;
4331        for (jump_marker = old_path.length - 1;
4332           (jump_marker > marker) && !jump_flag;
4333           jump_marker--) {
4334        //       cout << "Checking (" << old_path.x[ jump_marker] << ", "
4335        //          << old_path.y[ jump_marker] << ") <" << jump_marker << ">"
4336  << endl;
4337          if ((old_path.x[ jump_marker]  - old_path.x[ marker] >= -1) &&
4338            (old_path.x[ jump_marker]  - old_path.x[ marker] <= 1) &&
4339            (old_path.y[ jump_marker]  - old_path.y[ marker] >= -1) &&
4340            (old_path.y[ jump_marker]  - old_path.y[ marker] <= 1)) {
4341
```

```
4342          path_add(new_path, old_path.x[ jump_marker] ,
4343   old_path.y[ jump_marker] );
4344          //    cout << "Jumping from (" << new_path.x[ new_path.length - 2]
4345   << ", "
4346          //        << new_path.y[ new_path.length - 2] << ") to ("
4347          //        << new_path.x[ new_path.length - 1] << ", "
4348          //        << new_path.y[ new_path.length - 1] << ")" << endl;
4349
4350          marker = jump_marker;
4351          jump_flag = 1;
4352          }
4353        }
4354     }
4355   }
4356
4357   void agent::generate_world_path(path grid_path,      // Path in nav
4358   grid
4359                         path &world_path) // Path in world coords
4360   {
4361     // Convert path in grid cell coords to world coords
4362
4363     double wx, wy;              // World coords
4364     double xscale, yscale, zscale;    // Cell dimensions (tenths of
4365   inches)
4366     int xsize, ysize, zsize;         // Grid dimensions (# cells)
4367     int i;
4368
4369     xsize = nav_grid.msize[ 0] ;
4370     ysize = nav_grid.msize[ 1] ;
4371     zsize = nav_grid.msize[ 2] ;
4372
4373     xscale = (nav_grid.himv[ 0]  - nav_grid.lomv[ 0] ) * 120.0 / (double)
4374   xsize;
4375     yscale = (nav_grid.himv[ 1]  - nav_grid.lomv[ 1] ) * 120.0 / (double)
4376   ysize;
4377     zscale = (nav_grid.himv[ 2]  - nav_grid.lomv[ 2] ) * 120.0 / (double)
4378   zsize;
4379
4380     path_init(world_path);
4381
4382     for (i = 0; i < grid_path.length; i++) {
4383         wx = (((double) grid_path.x[ i]  + 0.5) * xscale
4384             + nav_grid.lomv[ 0]  * 120.0);
4385         wy = (((double) grid_path.y[ i]  + 0.5) * yscale
4386             + nav_grid.lomv[ 1]  * 120.0);
4387
4388         path_add(world_path, (int) wx, (int) wy);
4389     }
4390   }
4391
4392   void agent::path_init(path &p)      // Path
4393   {
4394     // Initialize path
4395
4396     p.length = 0;
4397   }
4398
4399   void agent::path_add(path &p, // Path
```

```
4400                        int x, int y)         // Point to add to path
4401   {
4402      // Add point to path
4403
4404      if (p.length == MAX_PATH_LENGTH) {
4405         cout << "path_add: Too many steps (> " << MAX_PATH_LENGTH << ")." <<
4406   endl;
4407         exit(-1);
4408      }
4409
4410      p.x[ p.length] = x;
4411      p.y[ p.length] = y;
4412      p.length++;
4413   }
4414
4415   int agent::check_clear(int x, int y)
4416   {
4417      //  Check to see whether region around point is free of known
4418   obstacles
4419
4420      int obs_count = 0;          // Obstacle counter
4421      int xi, yi;                 // Grid indices
4422      int xl, xh, yl, yh, zl, zh; // Grid coords of robot-sized box around
4423   point
4424      double wx, wy, wz;          // World coordinates of point
4425      double wxl, wxh, wyl, wyh;  // World coords of robot-sized box around
4426   point
4427
4428      //  int xsize, ysize;       // Grid dimensions (# cells)
4429      //  double xscale, yscale;  // Cell dimensions (tenths of inches)
4430      //  double xd, yd;          // Display coords
4431
4432      grid2world(nav_grid, x, y, 0, &wx, &wy, &wz);
4433
4434      wxl = wx - ROBOT_PASSAGE_RADIUS;
4435      wxh = wx + ROBOT_PASSAGE_RADIUS;
4436      wyl = wy - ROBOT_PASSAGE_RADIUS;
4437      wyh = wy + ROBOT_PASSAGE_RADIUS;
4438
4439      world2grid(nav_grid, wxl, wyl, wz, &xl, &yl, &zl);
4440      world2grid(nav_grid, wxh, wyh, wz, &xh, &yh, &zh);
4441
4442      //  xsize = NAV_X_RES;
4443      //  ysize = NAV_Y_RES;
4444
4445      //  xscale = (NAV_X_MAX - NAV_X_MIN) * 120.0 / (double) xsize;
4446      //  yscale = (NAV_Y_MAX - NAV_Y_MIN) * 120.0 / (double) ysize;
4447
4448      for (xi = xl; xi <= xh; xi++) {
4449        for (yi = yl; yi <= yh; yi++) {
4450          //       xd = (double) (xi + 0.5) * xscale + GLOBAL_X_MIN * 120.0;
4451          //       yd = (double) (yi + 0.5) * yscale + GLOBAL_Y_MIN * 120.0;
4452
4453          //       global_window->set_color("gold");
4454          //       global_window->display_point(xd, yd);
4455          //       global_window->set_color("black");
4456
4457          if (nav_grid.mapm[ grid2index(nav_grid, xi, yi, 0)] > 0) {
```

```
4458          //     global_window->set_color("red");
4459          //     global_window->display_circle(xd, yd, xscale);
4460          //     global_window->set_color("black");
4461          obs_count++;
4462        }
4463      }
4464    }
4465
4466    if (obs_count > MAX_OBS_COUNT) {
4467      return(0);
4468    }
4469    else {
4470      return(1);
4471    }
4472  }
4473
4474  int agent::check_frontier_arrival(int x, int y, int front_index)
4475  {
4476    //  Check to see whether region around point overlaps frontier
4477
4478    int xi, yi;                // Grid indices
4479    int xl, xh, yl, yh, zl, zh; // Grid coords of robot-sized box around
4480  point
4481    double wx, wy, wz;         // World coordinates of point
4482    double wxl, wxh, wyl, wyh; // World coords of robot-sized box around
4483  point
4484
4485    grid2world(nav_grid, x, y, 0, &wx, &wy, &wz);
4486
4487    wxl = wx - ROBOT_PASSAGE_RADIUS;
4488    wxh = wx + ROBOT_PASSAGE_RADIUS;
4489    wyl = wy - ROBOT_PASSAGE_RADIUS;
4490    wyh = wy + ROBOT_PASSAGE_RADIUS;
4491
4492    world2grid(nav_grid, wxl, wyl, wz, &xl, &yl, &zl);
4493    world2grid(nav_grid, wxh, wyh, wz, &xh, &yh, &zh);
4494
4495    for (xi = xl; xi <= xh; xi++) {
4496      for (yi = yl; yi <= yh; yi++) {
4497        if (check_frontier_cell(xi, yi, front_index)) {
4498        return(1);
4499        }
4500      }
4501    }
4502    return(0);
4503  }
4504
4505  double agent::closest_waypoint(path p,         // Path
4506                                 int x, int y,   // Current position (1/10
4507  inch)
4508                                 int index, // Index of current waypoint
4509                                 int &close_index) // Index of closest waypoint
4510  {
4511    // Finds waypoint furthest on path within destination tolerance, or
4512    // waypoint on path <p> closest to (x, y), returning the distance
4513  (inches)
4514    // to that point, and the waypoint's index in <index>
4515
```

```
4516      double dist;                    // Distance to waypoint
4517      double min_dist = MAX_DIST; // Minimum distance to waypoint
4518      int last_waypoint;              // Last waypoint to check
4519      int i;
4520
4521      //  cout << "current position = (" << x << ", " << y << ")" << endl;
4522
4523      if ((index < 0) || (index >= p.length)) {
4524        cout << "closest_waypoint: index <" << index << "> out of range
4525   [ 0.."
4526          << p.length << "]" << endl;
4527        exit(-1);
4528      }
4529
4530      // Set lookahead window for checking waypoints
4531
4532      last_waypoint = index + WAYPOINT_WINDOW;
4533      if (last_waypoint > p.length - 1) {
4534        last_waypoint = p.length - 1;
4535      }
4536
4537      // Search for closest waypoint
4538
4539      for (i = last_waypoint; i >= index; i--) {
4540        dist = hypot((double) (p.x[ i] - x), (double) (p.y[ i] - y)) / 10.0;
4541
4542        //    cout << "distance to waypoint [" << i << "] (" << p.x[ i] << ",
4543   "
4544        //         << p.y[ i] << ") = " << dist << endl;
4545
4546        if (dist < min_dist) {
4547          min_dist = dist;
4548          close_index = i;
4549
4550          if (min_dist <= LOCAL_NAV_TOLERANCE) {
4551          cout << "[ * ARRIVED *]" << endl;
4552          return(min_dist);
4553          }
4554        }
4555      }
4556
4557      //  cout << "closest waypoint [" << close_index << "] (" <<
4558   p.x[ close_index]
4559      //         << " , " << p.y[ close_index] << ") : dist = " << min_dist <<
4560   endl;
4561
4562      return(min_dist);
4563    }
4564
4565
4566    /******************* CORRIDOR FUNCTIONS *******************/
4567
4568    void agent::detect_corridors(void)
4569    {
4570      // Detect freespace cooridors in vicinity of robot
4571
4572      int i;
4573
```

```
4574        update();
4575
4576        for (i = 0; i < NUM_RANGE; i++) {
4577          corridor[ i] = check_corridor(i, CORRIDOR_FWD_RANGE,
4578                          CORRIDOR_SIDE_CLEARANCE);
4579          wide_corridor[ i] = check_corridor(i, CORRIDOR_WIDE_FWD_RANGE,
4580                              CORRIDOR_WIDE_SIDE_CLEARANCE);
4581        }
4582
4583        //  display_corridors();
4584        //  global_window->flush();
4585
4586        //  display_corridors();
4587      }
4588
4589
4590      int agent::check_corridor(int center,      // Index of sensor in center
4591      of corridor
4592                                int fwd_range,  // Required forward space
4593                                int side_clear) // Required side space
4594      {
4595        // Check whether a corridor exists centered around sensor <center>
4596        // Return 1 if true, 0 otherwise
4597
4598        int sensor;           // Sensor index
4599        int start;            // First sensor to be checked
4600        int end;          // Last sensor to be checked
4601
4602      // cout << "Checking corridor [" << center << "]" << endl;   // TEMP FIX
4603      for SCOUT
4604
4605        start = wrap(center - CORRIDOR_SPAN, 0, NUM_RANGE - 1);
4606        end = wrap(center + CORRIDOR_SPAN, 0, NUM_RANGE - 1);
4607
4608        if (start < end) {
4609          for (sensor = start; sensor <= end; sensor++) {
4610            if (!corridor_check_sensor(center, sensor, fwd_range, side_clear))
4611      {
4612      // cout << "Corridor [" << center << "] is >> BLOCKED <<." << endl;   //
4613      TEMP FIX for SCOUT
4614            return(0);
4615            }
4616          }
4617      // cout << "Corridor [" << center << "] is [[ OPEN ]]." << endl;    //
4618      TEMP FIX for SCOUT
4619          return(1);
4620        }
4621
4622        for (sensor = start; sensor < NUM_RANGE; sensor++) {
4623          if (!corridor_check_sensor(center, sensor, fwd_range, side_clear)) {
4624      // cout << "Corridor [" << center << "] is >> BLOCKED <<." << endl; //
4625      TEMP FIX for SCOUT
4626            return(0);
4627          }
4628        }
4629
4630        for (sensor = 0; sensor <= end; sensor++) {
4631          if (!corridor_check_sensor(center, sensor, fwd_range, side_clear)) {
```

```
4632     // cout << "Corridor [" << center << "] is >> BLOCKED <<." << endl;   //
4633     TEMP FIX for SCOUT
4634           return(0);
4635         }
4636     }
4637     // cout << "Corridor [" << center << "] is [[ OPEN ]]." << endl;   //
4638     TEMP FIX for SCOUT
4639       return(1);
4640     }
4641
4642     int agent::corridor_check_sensor(int center,            // Center sensor
4643     index
4644                                      int sensor,            // Sensor index
4645                                      int fwd_range,         // Required fwd space
4646                                      int side_clear)  // Required side space
4647     {
4648       // Check whether <sensor> is clear for corridor <center>
4649
4650       const double bot_width = ROBOT_RADIUS * 12.0; // Robot width (inches)
4651       const double sens_sep = SENSOR_SEP * 0.1;     // Sensor separation
4652     (degrees)
4653
4654       double angle;                    // Sensor angle (degrees)
4655       double center_angle;             // Angle of center sensor (degrees)
4656       double theta;                    // Angle (degrees) between sensor and
4657                             // perpendicular to center angle
4658       double thresh;         // Minimum clear distance (inches)
4659
4660       center_angle = angle_wrap((double) r.theta * 0.1
4661                            + (double) center * sens_sep);
4662     // TEMP FIX for SCOUT - changed r.turret to r.theta on previous line
4663     // cout << "center [" << center << "] : center angle = " << center_angle
4664     << endl;   // TEMP FIX for SCOUT
4665
4666       angle = angle_wrap((double) r.theta * 0.1 + (double) sensor *
4667     sens_sep);   // TEMP FIX for SCOUT r.turret to r.theta
4668       theta = 90.0 - angle_diff(center_angle, angle);
4669
4670       if (center == sensor) {
4671         thresh = fwd_range;
4672       }
4673       else {
4674         thresh = (bot_width + side_clear) / cos(theta * DEG2RAD)
4675           - bot_width;
4676         if (thresh > fwd_range) {
4677           thresh = fwd_range;
4678         }
4679       }
4680
4681     // cout << "sensor [" << sensor << "] : sensor angle = " << angle
4682     //         << " : theta = " << theta << " : thresh = " << thresh
4683     //         << " : range = " << r.range[ sensor];   // TEMP FIX for SCOUT
4684
4685       if (r.range[ sensor] < thresh) {
4686     // cout << " * BLOCKED *" << endl;   // TEMP FIX for SCOUT
4687         return(0);
4688       }
4689       else {
```

```
4690     // cout << " [ CLEAR ]" << endl;    // TEMP FIX for SCOUT
4691        return(1);
4692     }
4693  }
4694
4695  void agent::display_corridors(void)
4696  {
4697     // Display corridors in robot window
4698
4699     int i;
4700
4701     //  refresh_all();
4702
4703     for (i = 0; i < NUM_RANGE; i++) {
4704        if (wide_corridor[i] == 1) {
4705           display_corridor(global_window, i, CORRIDOR_WIDE_FWD_RANGE,
4706                     CORRIDOR_WIDE_SIDE_CLEARANCE, CORRIDOR_WIDE_COLOR);
4707           //      display_corridor_robot(i, CORRIDOR_WIDE_FWD_RANGE,
4708           //                      CORRIDOR_WIDE_SIDE_CLEARANCE,
4709           //                      CORRIDOR_WIDE_COLOR_ROBOT);
4710        }
4711        else ·if (corridor[i] == 1) {
4712           display_corridor(global_window, i, CORRIDOR_FWD_RANGE,
4713                     CORRIDOR_SIDE_CLEARANCE, CORRIDOR_COLOR);
4714           //      display_corridor_robot(i, CORRIDOR_FWD_RANGE,
4715           //                      CORRIDOR_SIDE_CLEARANCE,
4716  CORRIDOR_COLOR_ROBOT);
4717        }
4718     }
4719  }
4720
4721  void agent::display_corridor(window *win, // Window
4722                               int center,  // Center sensor index
4723                               int fwd_range,    // Required forward space
4724                               int side_clear,   // Required side space
4725                               char *color) // Corridor color
4726  {
4727     // Display corridor boundaries centered around sensor <center>
4728
4729     // Robot width (1/10 inch)
4730     const double bot_width = ROBOT_RADIUS * 120.0;
4731
4732     double fwd_dist;       // Length of forward axis (1/10 inch)
4733     double side_dist;      // Distance to either side of robot (1/10 inch)
4734     double angle;                         // Corridor angle (degrees)
4735     double x1, y1, x2, y2, x3, y3, x4, y4;  // Corner coords (1/10 inch)
4736     double fwd_x, fwd_y;                  // Offset for forward end
4737
4738     fwd_dist = bot_width + (double) fwd_range * 10.0;
4739     side_dist = bot_width + (double) side_clear * 10.0;
4740
4741  // SCOUT THESIS CHANGE - changed r.turret to r.theta in line below
4742     angle = angle_wrap((double) r.theta * 0.1
4743                     + (double) (center * SENSOR_SEP) * 0.1);
4744
4745     x1 = r.x + side_dist * cos((angle + 90.0) * DEG2RAD);
4746     y1 = r.y + side_dist * sin((angle + 90.0) * DEG2RAD);
4747
```

294

```
4748        x2 = r.x + side_dist * cos((angle - 90.0) * DEG2RAD);
4749        y2 = r.y + side_dist * sin((angle - 90.0) * DEG2RAD);
4750
4751        fwd_x = fwd_dist * cos(angle * DEG2RAD);
4752        fwd_y = fwd_dist * sin(angle * DEG2RAD);
4753
4754        x3 = x1 + fwd_x;
4755        y3 = y1 + fwd_y;
4756
4757        x4 = x2 + fwd_x;
4758        y4 = y2 + fwd_y;
4759
4760        win->set_color(color);
4761
4762        win->display_line(x1, y1, x2, y2);
4763        win->display_line(x2, y2, x4, y4);
4764        win->display_line(x4, y4, x3, y3);
4765        win->display_line(x3, y3, x1, y1);
4766
4767        win->set_color("black");
4768    }
4769
4770    void agent::display_corridor_robot(int center,   // Center sensor index
4771                                       int fwd_range,  // Required forward space
4772                                       int side_clear, // Required side space
4773                                       int color)      // Corridor color
4774    {
4775        // Display corridor boundaries centered around sensor <center> in
4776    robot window
4777
4778        // Robot width (1/10 inch)
4779        const double bot_width = ROBOT_RADIUS * 120.0;
4780
4781        double fwd_dist;        // Length of forward axis (1/10 inch)
4782        double side_dist;       // Distance to either side of robot (1/10 inch)
4783        double angle;                           // Corridor angle (degrees)
4784        double x1, y1, x2, y2, x3, y3, x4, y4;  // Corner coords (1/10 inch)
4785        double fwd_x, fwd_y;                    // Offset for forward end
4786
4787        fwd_dist = bot_width + (double) fwd_range * 10.0;
4788        side_dist = bot_width + (double) side_clear * 10.0;
4789
4790    // SCOUT THESIS CHANGE - changed r.turret to r.theta in line below
4791        angle = angle_wrap((double) r.theta * 0.1
4792                        + (double) (center * SENSOR_SEP) * 0.1);
4793
4794        x1 = r.x + side_dist * cos((angle + 90.0) * DEG2RAD);
4795        y1 = r.y + side_dist * sin((angle + 90.0) * DEG2RAD);
4796
4797        x2 = r.x + side_dist * cos((angle - 90.0) * DEG2RAD);
4798        y2 = r.y + side_dist * sin((angle - 90.0) * DEG2RAD);
4799
4800        fwd_x = fwd_dist * cos(angle * DEG2RAD);
4801        fwd_y = fwd_dist * sin(angle * DEG2RAD);
4802
4803        x3 = x1 + fwd_x;
4804        y3 = y1 + fwd_y;
4805
```

```
4806        x4 = x2 + fwd_x;
4807        y4 = y2 + fwd_y;
4808
4809        draw_line((int) x1, (int) y1, (int) x2, (int) y2, color + 2);
4810        draw_line((int) x2, (int) y2, (int) x4, (int) y4, color + 2);
4811        draw_line((int) x4, (int) y4, (int) x3, (int) y3, color + 2);
4812        draw_line((int) x3, (int) y3, (int) x1, (int) y1, color + 2);
4813    }
4814
4815    int agent::select_corridor(double heading)        // Heading (degrees)
4816    {
4817        // Select corridor nearest to specified heading
4818
4819        const double sens_sep = SENSOR_SEP * 0.1;        // Sensor separation
4820    (degrees)
4821        double angle;                          // Sensor angle
4822        double dtheta;                // Angle/heading deviation
4823        double min_dtheta = 360.0;          // Minimum angle deviation
4824        int select = -1;                  // Index of selected corridor
4825        int i;
4826
4827        heading = angle_wrap(heading);
4828
4829        for (i = 0; i < NUM_RANGE; i++) {
4830            if (corridor[ i] == 1) {
4831                angle = angle_wrap((double) r.theta * 0.1
4832                            + (double) i * sens_sep);
4833    // SCOUT THESIS CHANGE - use r.theta vice r.turret in line above
4834    // TEMP FIX for SCOUT - lets try some numbers checking below
4835    // cout << "About to call angle_diff with heading= " << heading
4836    //         << "and angle = " << angle << endl;    // TEMP FIX for SCOUT
4837            dtheta = angle_diff(heading, angle);
4838    // cout << "dtheta = angle_diff(heading,angle) = " << dtheta << endl;
4839    // TEMP FIX for SCOUT
4840    // cout << "min_dtheta = " << min_dtheta << "  i = " << i << endl;    //
4841    TEMP FIX for SCOUT
4842            if (dtheta < min_dtheta) {
4843            min_dtheta = dtheta;
4844            select = i;
4845            }
4846        }
4847    }
4848
4849        if (select == -1) {
4850            cout << "No open corridors." << endl;
4851            return(select);
4852        }
4853    //SCOUT THESIS CHANGE -  changed r.turret to r.theta 3 lines down
4854        cout << "desired heading = " << heading << " : selected corridor ["
4855            << select << "] : corridor angle = "
4856            << angle_wrap((double) r.theta * 0.1 + (double) select *
4857    sens_sep)
4858            << " : deviation = " << min_dtheta << endl;
4859
4860        if (wide_corridor[ select] == 1) {
4861        //      display_corridor(global_window, select,
4862    CORRIDOR_WIDE_FWD_RANGE,
```

```
4863            //                    CORRIDOR_WIDE_SIDE_CLEARANCE,
4864    CORRIDOR_SELECT_WIDE_COLOR);
4865            //      display_corridor_robot(select, CORRIDOR_WIDE_FWD_RANGE,
4866            //                    CORRIDOR_WIDE_SIDE_CLEARANCE,
4867            //                    CORRIDOR_SELECT_WIDE_COLOR_ROBOT);
4868       }
4869       else {
4870            //      display_corridor(global_window, select, CORRIDOR_FWD_RANGE,
4871            //                    CORRIDOR_SIDE_CLEARANCE, CORRIDOR_SELECT_COLOR);
4872            //      display_corridor_robot(select, CORRIDOR_FWD_RANGE,
4873    CORRIDOR_SIDE_CLEARANCE,
4874            //                    CORRIDOR_SELECT_COLOR_ROBOT);
4875       }
4876
4877       return(select);
4878    }
4879
4880    /********** INTERFACE TO CONTINUOUS LOCALIZATION **********/
4881
4882    void agent::connect_cl(void)
4883    {
4884       // Initialize communications with continuous localization
4885
4886       char comm_mach[ STRLEN] ;
4887
4888       cout << "Enter continuous localization host ==> ";
4889       cin >> comm_mach;
4890       cin.get();
4891
4892       //   connect_to_CL(CONTLOC_CHANNEL, CONTLOC_HOST);
4893       //   cout << "Connected to CONTINUOUS LOCALIZATION on " << CONTLOC_HOST
4894    << "."
4895       //          << endl;
4896
4897       connect_to_CL(CONTLOC_CHANNEL, comm_mach);
4898       cout << "Connected to CONTINUOUS LOCALIZATION on " << comm_mach << "."
4899            << endl;
4900
4901       contloc_mode = 1;
4902    }
4903
4904    void agent::send_cl_grid(void)
4905    {
4906       // Send global grid to continuous localization
4907
4908       if (!contloc_mode) {
4909          return;
4910       }
4911
4912       cout << "Sending global grid to CONTINUOUS LOCALIZATION." << endl;
4913       save_grid_file(global_grid, ARIEL_CL_FILE, "");
4914
4915    // SCOUT THESIS CHANGE - if continuouse localization is ever used send
4916    r.theta instead of r.turret
4917       sendroom_to_CL(CONTLOC_CHANNEL, ARIEL_CL_FILE, (double) r.x / 10.0,
4918                   (double) r.y / 10.0, (double) r.theta / 10.0,
4919                   (double) r.theta / 10.0, 0, 0.0, 0.0, 0.0);
4920    }
```

```
4921
4922    /********* MULTIROBOT COMMUNICATION *********/
4923
4924    // BEGIN SCOUT THESIS CHANGE
4925    // This routine now sets up communication for up to MAX_ROBOTS
4926    simultaneously
4927    //  2 robot limitation is eliminated
4928
4929    void agent::init_robot_comm(void)
4930    {
4931      // Initialize robot communication channel
4932
4933      char robot_server_name [STRLEN];  // Server robot host name
4934
4935      // If Server Robot
4936      if (r.id == SERVER_ROBOT) {
4937        if (init_comm_server(ARIEL_CHANNEL, PORT_ARIEL, NONBLOCK_COMM) == 0)
4938    {
4939          cout << "init_robot_comm: Robot [" << r.id
4940            << "] initialized communications as server." << endl;
4941          multi_mode = 1;
4942        }
4943        else {
4944          cout << "init_robot_comm: Robot [" << r.id
4945            << "] unable to set up communications as server." << endl;
4946          multi_mode = 0;
4947        }
4948      }
4949      else if (r.id <= MAX_ROBOTS) {
4950        cout << "Enter host name for server robot  ==> ";
4951        cin >> robot_server_name;
4952
4953        if (init_comm_client(ARIEL_CHANNEL, robot_server_name,
4954          BASE_CLIENT_PORT + r.id, NONBLOCK_COMM) == 0) {
4955          cout << "init_robot_comm: Robot [" << r.id
4956            << "] initialized communications as client." << endl;
4957          multi_mode = 1;
4958        }
4959        else {
4960          cout << "init_robot_comm: Robot [" << r.id
4961            << "] unable to set up communications as client." << endl;
4962          multi_mode = 0;
4963        }
4964      }
4965      else {
4966        cout << "init_robot_comm: Robot [" << r.id
4967          << "] unable to set up communications for more than "
4968          << MAX_ROBOTS
4969          << "  robots." << endl;
4970        multi_mode = 0;
4971      }
4972    }
4973    //END SCOUT THESIS CHANGE
4974
4975    void agent::send_robot_message(char *message)
4976    {
4977      // Send message to other robot
4978    // BEGIN SCOUT THESIS CHANGE
```

```
4979        cout << "Sending message <" << message << ">." << endl;
4980        // Loop thru all possible client robots connections, send message
4981        // that new map is available.
4982        for (int i=1; i< MAX_ROBOTS; i++){
4983            write_comm(i, message, strlen(message) + 1);
4984        }
4985    //END SCOUT THESIS CHANGE
4986    }
4987
4988    void agent::user_send_robot_message(void)
4989    {
4990        // Send message to other robot (user mode)
4991
4992        char message[ STRLEN] ;
4993
4994        cout << "Enter message ==> ";
4995        cin >> message;
4996
4997        cout << "Sending message <" << message << ">." << endl;
4998
4999        write_comm(ARIEL_CHANNEL, message, strlen(message) + 1);
5000    }
5001
5002    //BEGIN SCOUT THESIS CHANGE
5003    // Pass in the channel used for communication between client
5004    // and server
5005
5006    int agent::receive_robot_message(int channel, char *message)
5007    {
5008        // Receive message from other robot
5009        // Returns 1 if message received, 0 otherwise
5010
5011        int message_received;        // Message receipt flag
5012
5013        message_received = read_comm(channel, message, STRLEN);
5014    // END SCOUT THESIS CHANGE
5015        if (message_received) {
5016            cout << "Received message <" << message << ">." << endl;
5017        }
5018
5019        return(message_received);
5020    }
5021
5022    void agent::user_receive_robot_message(void)
5023    {
5024        // Receive message from other robot (user mode)
5025
5026        char message[ STRLEN] ;
5027
5028        if (read_comm(ARIEL_CHANNEL, message, STRLEN) == 0) {
5029            cout << "No messages waiting." << endl;
5030        }
5031        else {
5032            cout << "Received message <" << message << ">." << endl;
5033        }
5034    }
5035
5036    /******************** MULTIROBOT EXPLORATION ********************/
```

299

```
5037
5038    void agent::integrate_remote_map(void)
5039    {
5040       // Integrate new map from remote robot (if a new map exists)
5041
5042       Map3D remote_grid;            // Evidence grid for remote map
5043       char mapfile[ STRLEN] ;       // Remote map file
5044       char posinfo[ STRLEN] ;       // Remote map position information
5045       int mx, my, mtheta;           // Position of center of new map
5046                        // (1/10 inch, 1/10 degree)
5047    // BEGIN SCOUT THESIS CHANGE
5048       int channel;                  // Channel number
5049
5050       // Loop thru all channels corresponding to client robots. Check each
5051       // channel to see if we received a new map message.
5052       // Robot 2 is on channel 1, Robot3 is on channel 2, . . .
5053
5054       for (channel=1; channel < MAX_ROBOTS; channel++){
5055
5056          // Check for new map message
5057
5058          if (!receive_robot_message(channel, mapfile)) {
5059             continue;                // If nothing to read on this channel,
5060                        // do not give up, continue will jump
5061                        // back to "for" loop and increment
5062                        // channel counter.
5063          }
5064
5065          cout << "New map from remote robot in <" << mapfile << ">." <<
5066    endl;
5067
5068          // Load grid along with position info
5069
5070          if (!load_grid_file_com(&remote_grid, mapfile, posinfo)) {
5071             return;
5072          }
5073
5074          sscanf(posinfo, "%d %d %d", &mx, &my, &mtheta);
5075
5076          cout << "New map position = (" << mx << ", " << my << ") [" <<
5077    mtheta << "]"
5078                 << endl;
5079
5080          // Display and integrate new map
5081
5082          //  grid_display(grid_window, remote_grid);
5083
5084    // NEW MAJOR SCOUT THESIS change below
5085    // if r.id==1 then robot is SERVER and needs to integrate a local scan
5086    to the global map
5087    // if r.id !=1 then robot is CLIENt and needs to intgrate the SERVER
5088    global map that is sent
5089
5090       if (r.id==1)  {
5091          integrate_grid(global_grid, remote_grid, (double) mx / 120.0,
5092                        (double) my / 120.0,  (double) mtheta / 10.0);
5093       }
5094       else  {
```

```
5095
5096        integrate_global_grid(global_grid, remote_grid, (double) mx /
5097    120.0,
5098                        (double) my / 120.0, (double) mtheta / 10.0);
5099      }      // close for else r.id !=1
5100
5101      grid_display_global(global_grid);
5102
5103      }      // close for channel check counter
5104    // END SCOUT THESIS CHANGE
5105    }
```

APPENDIX I. FRONTIER-BASED EXPLORATION CODE – COMM.H

This appendix contains the header file for the communications routine that allows multiple robots to send messages to one another.

```
1   /* Original code written by William Adams */
2
3   /*  Modifications for increased number of robots June 1998
4       for Master's Thesis work by Patrick A. Hillmeyer        */
5
6   /* include file for comm.c and all files linking to comm.o */
7
8   enum { OFF_COMM, NONBLOCK_COMM, BLOCK_COMM} ;
9
10  #define PORT_DETECT 65003
11  #define PORT_GESTURE 65004
12  #define PORT_AUXPORT 65005
13  #define PORT_CONTSERV 65006
14  #define PORT_SEARCH 65007
15  #define PORT_CONTLOC 65008
16  #define PORT_ARIEL 65009
17
18  /* BEGIN SCOUT THESIS CHANGE */
19
20  #define MAX_ROBOTS 9   /* This is the max number of robots that can
21                            operate at one time  - change as you get
22                            more robots  -  this must always be one less
23                            than MAXCHANNEL  - see important note about
24                            MAXCHANNEL and comm.c file.            */
25
26  #define SERVER_ROBOT 1 /* This is the ID number (in Nserver) of the
27                            robot that will act as the Server robot
28                            to the other Client robots for receiving and
29                            sending .eg files */
30
31  #define BASE_CLIENT_PORT 65007
32
33
34
35  /*  IMPORTANT!!  The initialization of global arrays sd, ld, and
36  comm_mode in the file comm.c has to match EXACTLY with MAXCHANNEL.
37  Example: if MAXCHANNEL is 10 then you need 10 zeros to initialize each
38  of the arrays mentioned above.              */
39
40  #define MAXCHANNEL  MAX_ROBOTS + 1
41                  /* Any single process can communicate with this
42                     many other processes.  Although the port numbers
43                     on both ends of the communication must agree,
44                     the channel numbers do not need to agree.
45                     Within a single process, each communication
46                     link must have a unique channel number. */
47                  /* If this is changed, you must change the initializing
48                     declaration using it in comm.c */
49
50  /* END SCOUT THESIS CHANGE */
```

303

APPENDIX J. FRONTIER-BASED EXPLORATION CODE – COMM.C

This appendix contains the source code for the communications routine that allows multiple robots to send messages to one another.

```
1    /*********************************************************************
2     * comm_server.c
3     * written: 11/22/95  William Adams
4     * last modifed: 1/22/95 William Adams
5     *
6     * Set up internet communication on a single port.
7     * Receive requests from ONE other process and send back
8     * information.
9
10    *********************************************************************/
11
12   /*  Modifications for increased number of robots June 1998
13        for Master's Thesis work by Patrick A. Hillmeyer      */
14
15   #include <stdio.h>
16   #include <sys/types.h>
17   #include <sys/socket.h>
18   #include <netinet/in.h>
19   #include <netdb.h>
20   #include <fcntl.h>
21   #include <errno.h>
22
23   #include "comm.h"
24
25   enum status { NOTHING_C, HALFWAY_C, READY_C } ;
26
27   // BEGIN SCOUT THESIS CHANGE
28   // *** see notes in comm.h file concerning these next few lines
29
30   int sd[ MAXCHANNEL]   = { 0,0,0,0,0,0,0,0,0,0} ;   /* socket handle */
31   int ld[ MAXCHANNEL]   = { 0,0,0,0,0,0,0,0,0,0} ;
32   int comm_mode[ MAXCHANNEL]  = { 0,0,0,0,0,0,0,0,0,0} ;
33
34   // END SCOUT THESIS CHANGE
35
36   int haveaclient = 0;
37
38   /* wait client to call, blocking */
39   int comm_wait_for_client(channel,control
40   int channel,control;
41   {
42      if (comm_mode[ channel]  == NOTHING_C) {
43         fprintf(stderr,
44           "\nImproper call to comm_wait_for_client...use
45   init_comm_server.\n" );
46         return(5);
47      }
48      else if (comm_mode[ channel]  == READY_C) {
49         fprintf(stderr,
50           "\nRedundant call to comm_wait_for_client, ignored\n" );
51         return(0);
52      }
```

305

```
53
54        /* else comm_mode[ channel]  == HALFWAY_C which is correct */
55
56        if ((sd[ channel] =accept(ld[ channel] ,0,0))<0) {
57          perror("INET Domain Accept");
58          return(5);
59        }
60
61          /* set to non-blocking if specified, else default is blocking */
62        if (control==NONBLOCK_COMM)
63          fcntl(sd[ channel] ,F_SETFL,O_NDELAY);
64
65        comm_mode[ channel] = READY_C;  /* success */
66
67        return(0);  /* success */
68     }
69
70
71
72     int init_comm_server(channel,port_num,control)
73     int channel,port_num,control;
74     {
75     // BEGIN SCOUT THESIS CHANGE
76        static int num_socs = 0; /* Number of sockets already established. */
77        int rc;
78        int addrlen;
79        struct sockaddr_in name;
80        struct sockaddr_in *ptr;
81        struct sockaddr addr;
82        struct hostent *hp, *gethostbyaddr();
83        int err;
84
85        // If you are the SERVER ROBOT, set up next available channel
86        // for the new client robot
87        if (port_num == PORT_ARIEL){
88            channel = ++num_socs;
89            port_num = BASE_CLIENT_PORT + num_socs + 1;
90      }
91     // END SCOUT THESIS CHANGE
92
93        /* create a "listen" socket to receive service requests */
94        if ((ld[ channel] =socket(AF_INET,SOCK_STREAM,6))<0) {
95          perror("INET Domain Socket");
96          return(1);
97        }
98
99        /* initialize fields in an Internet address structure */
100       name.sin_family = (short int) AF_INET;
101       name.sin_port = htons(port_num);
102       name.sin_addr.s_addr = INADDR_ANY;
103
104       /* bind the Internet address to the Internet socket */
105       if (bind(ld[ channel] ,(struct sockaddr *)&name,sizeof(name))<0) {
106         close(ld[ channel] );
107         perror("INET Domain Bind");
108         return(2);
109       }
110
```

```
111        /* find out the port number assigned to our socket */
112        addrlen = sizeof(addr);
113        if ((rc=getsockname(ld[channel],&addr,&addrlen))<0) {
114          perror("INET Domain getsockname");
115          return(3);
116        }
117
118        /* now "advertise" the port number assigned to us */
119        ptr = (struct sockaddr_in *) &addr;
120  /*   printf("\n\tSocket has port number: %d\n",ntohs(ptr->sin_port));   */
121
122        /* mark socket as a passive "listen" socket */
123        if (listen(ld[channel],5)<0) {
124          perror("INET Domain Listen");
125          return(4);
126        }
127
128        /* wait for a client to contact us... (blocking) */
129        comm_mode[channel] = HALFWAY_C;
130        if ((err=comm_wait_for_client(channel,control)) != 0) {
131          return(err);
132        }
133
134        /* find out who's calling us */
135  /*if ((rc=getpeername(sd[channel],&addr,&addrlen))<0) {
136          perror("INET Domain getpeername");
137          return(6);
138        }  */
139
140        /* "announce" the caller */
141  /*printf("\n\tCalling socket from host %s\n",inet_ntoa(ptr->sin_addr));
142      printf("\n\t   has port number %d\n",ptr->sin_port);
143      if ((hp=gethostbyaddr(&ptr->sin_addr,4, AF_INET)) != NULL) {
144        printf("\tFrom hostname: %s\n\tWith aliases:",hp->h_name);
145        while (*hp->h_aliases)
146          printf("\n\t\t\t%s",*hp->h_aliases++);
147        printf("\n\n");
148      }
149      else {
150        perror("\n\tgethostbyaddr() failed");
151        printf("\n\th_errno is %d\n\n",h_errno);
152      }  */
153
154      comm_mode[channel] = READY_C;
155      return(0);
156  }
157
158
159
160    int init_comm_client(channel,mach_name,port_num,control)
161    int channel;
162    char *mach_name;
163    int port_num,control;
164    {
165      struct sockaddr_in name;
166      struct hostent *hp, *gethostbyaddr();
167
168      /* create a "client" socket to request service */
```

```
169     if ((sd[ channel] =socket(AF_INET,SOCK_STREAM,0))<0) {
170       perror("INET Domain Socket");
171       return(1);
172     }
173
174     /* initialize fields in an Internet address structure */
175     name.sin_family = AF_INET;
176     name.sin_port = htons(port_num);
177     hp=gethostbyname(mach_name);
178     memcpy(&name.sin_addr.s_addr,hp->h_addr,hp->h_length);
179
180     if (connect(sd[ channel],(struct sockaddr *)&name,sizeof(name))<0) {
181       perror("Connect()");
182       return(2);
183     }
184
185       /* set to non-blocking if specified, else default is blocking */
186     if (control==NONBLOCK_COMM)
187       fcntl(sd[ channel],F_SETFL,O_NDELAY);
188
189     comm_mode[ channel] = READY_C;   /* success */
190     return(0);
191   }
192
193
194
195   int read_comm(channel,buf,bufsize)
196   int channel;
197   char *buf;
198   int bufsize;
199   {
200     int nbytes;
201
202   // BEGIN SCOUT CHANGE
203     // If no socket has been established on this channel,
204     // then return.
205     if (sd[ channel] == 0){
206         return(0);
207     }
208   // END SCOUNT CHANGE
209
210     if (comm_mode[ channel] == READY_C) {
211       memset(buf,0,bufsize);
212       if ((nbytes=read(sd[ channel],buf,bufsize))<0) {
213         if (errno!=EWOULDBLOCK) {
214         perror("INET domain Read");
215         return(-1);  /* indicate error */
216         }
217         else    /* it was just an unblocked read with no data ready */
218         return(0);
219       }
220       else if (nbytes==0) {
221         fprintf(stderr,"\nSender Disappeared.\n");
222         comm_mode[ channel] = HALFWAY_C;
223         return(-2);  /* no data to be read */
224       }
225       else {
226         return(nbytes);  /* read data */
```

```
227            }
228        }
229      else   /* socket has not been initialized */
230        return(-3);
231    }
232
233
234    void write_comm(channel,buf,bufsize)
235    int channel;
236    char *buf;
237    int bufsize;
238    {
239      if (comm_mode[ channel] == READY_C)
240        write(sd[ channel] ,buf,bufsize);
241    }
242
243
244
245    demoserver()      /* demo */
246    {
247      int a=0;
248      int b=10;
249      int c=100;
250      char buf[ 256] ;
251
252      init_comm_server(0,65003,NONBLOCK_COMM);
253      while (1) {
254        if (read_comm(0,buf,sizeof(buf))>0) {   /* if read something */
255          sprintf(buf,"%d %d %d\ 0",a++,b++,c++);
256          write_comm(0,buf,sizeof(buf));
257        }
258        sleep(1);
259      }
260    }
261
262
263    democlient()
264    {
265      int a,b,c;
266      char buf[ 256] ;
267
268      init_comm_client(0,"coyote",65003,BLOCK_COMM);
269      while (1) {
270        strcpy(buf,"Request");
271        write_comm(0,buf,sizeof(buf));
272        read_comm(0,buf,sizeof(buf));
273        sscanf(buf,"%d %d %d",&a,&b,&c);
274        printf("\nreceived %d %d %d\n",a,b,c);
275        fflush(stdout);
276        sleep(1);
277      }
278    }
```

LIST OF REFERENCES

1. General Krulak's comments to the AFCEA/U.S. Naval Institute West '98 Conference, "The Race Goes to the Swiftest In Commercial, Military Frays," *Signal*, March 1998.

2. Gage, D. W., "Cost-Optimization of Many-Robot Systems," *Proceedings of SPIE Mobile Robots IX*, Boston, MA, November 1994.

3. Alptekin, G., "Geometric Formation with Uniform Distribution and Movement In Formation of Distributed Mobile Robots," Master's Thesis, Naval Postgraduate School, June 1996.

4. *NOMAD 200 Hardware Manual*, Nomadic Technologies, Inc., Mountain View, CA, 1997.

5. *Sensus 300: Infrared Ranging System*, Nomadic Technologies, Inc., Mountain View, CA, 1997.

6. *Sensus 200: Sonar Ranging System*, Nomadic Technologies, Inc., Mountain View, CA, 1997.

7. *Sensus 500: Laser Ranging System*, Nomadic Technologies, Inc., Mountain View, CA, 1997.

8. Latt, K., "Sonar-based Localization of Mobile Robots using the Hough Transform," Master's Thesis, Naval Postgraduate School, March 1997.

9. Glennon, J., "Feature-Based Localization of an Autonomous Mobile Robot using the Hough Transform and an Unsupervised Learning Network," Master's Thesis, Naval Postgraduate School, June 1998.

10. *SCOUT Beta 1.1*, Nomadic Technologies, Inc., Mountain View, CA, 1998.

11. *NOMAD SCOUT*, Nomadic Technologies, Inc., Mountain View, CA, 1997.

12. *Nomadic Host Development Environment*, Nomadic Technologies, Inc., Mountain View, CA, 1997.

13. *Mercury-RF1-TCP User's Guide Version 1.7*, Nomadic Technologies, Inc., Mountain View, CA, 1997.

14. *RangeLAN2 Access Point Models 7510 and 7520 User's Guide*, Proxim, Inc., Mountain View, CA, 1997.

15. Bornstein, J., Everett, H. R., and Feng, L., *Navigating Mobile Robots: Systems and Techniques*, A K Peters, 1996.

16. Moravec, H. P. and Elfes, A., "High Resolution Maps from Wide Angle Sonar," *IEEE International Conference on Robotics and Automation*, St. Louis, MO, March 1985.

17. Martin, M. C. and Moravec, H. P., *Robot Evidence Grids*, CMU-RI-TR-96-06, Carnegie Mellon University, Pittsburgh, PA, March 1996.

18. Elfes, A., "Sonar-Based Real-World Mapping and Navigation," *IEEE Journal of Robotics and Automation*, Vol. RA-3, No. 3, pp. 249-265, 1987.

19. Moravec, H. P., "Sensor Fusion in Certainty Grids for Mobile Robots," *AI Magazine*, Vol. 9, No. 2, pp. 61-74, 1988.

20. Graves, K., Adams, W. and Schultz, A., "Continuous Localization in Changing Environments," *Proceedings of the 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation*, Monterey, CA, July 1997.

21. Moravec, H. P., "Certainty Grids for Mobile Robots," *Proceedings of the Workshop on Space Telerobotics*, Jet Propulsion Laboratory, Pasadena, CA, July 1987.

22. Yamauchi, B., Schultz, A. and Adams, W., "Mobile Robot Exploration and Map-Building with Continuous Localization," *Proceedings of the 1998 IEEE International Conference on Robotics and Automation*, Leuven, Belgium, May 1998.

23. Yamauchi, B., "A Frontier-Based Approach for Autonomous Exploration," *Proceedings of the 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation*, Monterey, CA, July 1997.

24. Klaus, B. and Horn, P., *Robot Vision*, The MIT Press, 1986.

25. Yamauchi, B., "Frontier-Based Exploration Using Multiple Robots," *Proceedings of the Second International Conference on Autonomous Agents*, Minneapolis, MN, May 1998.

26. Yun, X., EC 4300 Class Notes, Naval Postgraduate School, 1997.

27. MacKenzie, P. and Dudek, G., "Precise Positioning using Model-Based Maps," *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, San Diego, CA, May 1994.

28. Dudek, G., Jenkin, M., Milios, E. and Wilkes, D., *Reflections on Modelling a Sonar Range Sensor*, CIM-92-9, McGill University, Montréal, Québec, Canada, May 1996.

29. Gage, D. W., "Telerobotic Requirements for Sensing, Navigation, and Communications," *Proceedings of the 1994 IEEE National Telesystems Conference*, San Diego, CA, May 1994.

30. Yuta, S. and Premvuti, S., "Coordinating Autonomous and Centralized Decision Making to Achieve Cooperative Behaviors between Multiple Mobile Robots," *Proceedings of the 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Raleigh, NC, July 1992.

31. Gage, D. W., "Development and Command-Control Tools for Many Robot Systems," *Proceedings of SPIE Microrobotics and Micromechanical Systems*, Philadelphia, PA, April 1995.

32. Gage, D. W., "How to Communicate with Zillions of Robots," *Proceedings of SPIE Mobile Robots VIII*, Boston, MA, September 1993.

33. Gage, D. W., "Network Protocols for Mobile Robot Systems," *Proceedings of SPIE Mobile Robots XII*, Pittsburgh, PA, October 1997.

34. Yamauchi, B., "Mobile Robot Localization in Dynamic Environments Using Dead Reckoning and Evidence Grids," *Proceedings of the 1996 IEEE International Conference on Robotics and Automation*, Minneapolis, MN, April 1996.

35. Rekleitis, I., Dudek, G. and Milios, E., "Multi-Robot Exploration of an Unknown Environment, Efficiently Reducing the Odometry Error," *Proceedings of the International Joint Conference in Artificial Intelligence*, Nagoya, Japan, August 1997.

36. Singh, K. and Fujimura, K., "Map Making by Cooperating Mobile Robots," *Proceedings of the 1993 IEEE International Conference on Robotics and Automation*, Atlanta, GA, May 1993.

37. Gage, D. W., "Sensor Abstractions to Support Many-Robot Systems," *Proceedings of SPIE Mobile Robots VII*, Boston, MA, November 1992.

38. Jacobus, C. J., Mitchell, B. T., Jacobus, H. N., Watts, R. C., Taylor, M. J. and Salazar, A., "Man-Portable Command, Communication, and Control Systems for the Next Generation of Unmanned Field Systems," *Proceedings of SPIE Mobile Robots VII*, Boston, MA, November 1992.

39. Mays, E. J. and Reid, F. A., "Shepherd Rotary Vehicle: Multivariate Motion Control and Planning," Master's Thesis, Naval Postgraduate School, September 1997.

40. Leonardy, T., "Implementation and Evaluation of an INS System for the Shepherd Rotary Vehicle," Master's Thesis, Naval Postgraduate School, December 1997.

41. Wang, J., Premvuti, S. and Tabbara, A., "A Wireless Medium Access Protocol (CSMA/CD-W) for Mobile Robot based Distributed Robotics Systems," *Proceedings of the 1995 IEEE International Conference on Robotics and Automation*, Nagoya, Japan, May 1995.

313

42. Everett, H. R., Gage, D. W., Gilbreath, G. A., Laird, R. T. and Smurlo, R. P. "Real-World Issues in Warehouse Navigation," *Proceedings of SPIE Mobile Robots IX*, Boston, MA, November 1994.

BIBLIOGRAPHY

Durrant-Whyte, H. F., *Integration, Coordination and Control of Multi-Sensor Robot Systems*, Kluwer Academic Publishers, 1988.

Stevens, W. R., *TCP/IP Illustrated, Volume 3*, Addison-Wesley, 1996.

Wright, G. R., Stevens, W. R., *TCP/IP Illustrated, Volume 2*, Addison-Wesley, 1995.

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center...2
8725 John J. Kingman Rd., STE 0944
Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library..2
Naval Postgraduate School
411 Dyer Rd.
Monterey, CA 93943-5101

3. Director, Training and Education 1
MCCDC, Code C46
1019 Elliot Rd.
Quantico, VA 22134-5027

4. Director, Marine Corps Research Center..2
MCCDC, Code C40RC
2040 Broadway Street
Quantico, VA 22134-5107

5. Director, Studies and Analysis Division...1
MCCDC, Code C45
300 Russell Road
Quantico, VA 22134-5130

6. Marine Corps Representative...1
Naval Postgraduate School
Code 037, Bldg. 234, HA-220
699 Dyer Road
Monterey, CA 93940

7. Marine Corps Tactical Systems Support Activity.......................................1
Technical Advisory Branch
Attn: Maj J.C. Cummiskey
Camp Pendleton, CA 92055-5080

8. Douglas W. Gage...1
SPAWARSYSCEN D371
53406 Woodward Road
San Diego, CA 92152-7383